# The Things That Count: Coverage Evaluation Under the Microscope (Registered Report)

Tobias Holl*
Ruhr University Bochum
tobias.holl@rub.de

Leon Weiß*
Ruhr University Bochum
leon.weiss@rub.de

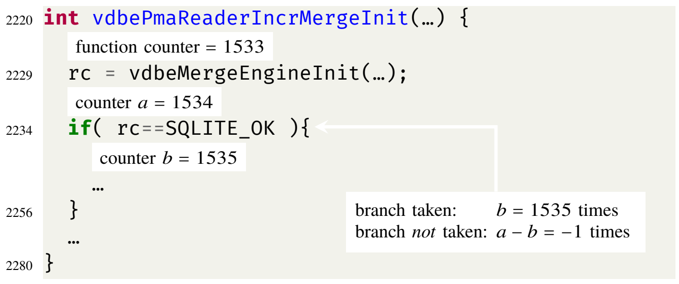
Kevin Borgolte
Ruhr University Bochum
kevin.borgolte@rub.de

*These authors contributed equally to this work.

***Abstract*—Current best practice recommendations for fuzzing research rely on the use of standardized evaluation frameworks. While many aspects of these frameworks have been scrutinized, the coverage collection and evaluation remain blind spots. We present two examples of discrepancies in coverage reported by FuzzBench, highlighting a gap in our understanding of the evaluation frameworks we depend on. In this work, we close this gap for FuzzBench by systematically examining its coverage analysis. We find that there are issues in every stage of the analysis pipeline that can potentially influence reported coverage.**

**We propose a thorough experimental design to assess the impact of each individual flaw, and their combined influence on the evaluation results. We will contextualize our results within the existing literature and discuss implications for our trust in previous fuzzing evaluations. With this work, we will improve confidence in the reliabilty of standardized fuzzing benchmarks and inform future research on how to improve their evaluation.**

**We will open-source our code after completing all experiments.**



```
int vdbePmaReaderIncrMergeInit(…) {
  function counter = 1533
  rc = vdbeMergeEngineInit(…);
  counter a = 1534
  if( rc==SQLITE_OK ){
    counter b = 1535
    …
  }
  …
}
```

branch taken: $b = 1535$ times
branch *not* taken: $a - b = -1$ times

Listing 1: Simplified race condition example from SQLite. The function counter and counter $a$ were not incremented correctly due to a data race. The calculation $a - b$ then underflows and evaluates to $2^{64} - 1$.

## I. Introduction

Fuzzing has become a highly effective technique for discovering software bugs and vulnerabilities. It continues to be an active research area for both academia and industry and, unsurprising given its success, has also been widely adopted by practitioners. Given the significant research activity in the field, there are hundreds of different fuzzing techniques [11]. To ensure a fair and rigorous comparison between all these competing fuzzers, researchers have long identified the need for standardized evaluation approaches and benchmarks [8].

This has led to the development of coverage-based benchmarks, like FuzzBench [12], which are now recommended for rigorous fuzzer evaluation [16]. FuzzBench itself provides infrastructure for comparing fuzzers' performance based on the code coverage that the fuzzers achieve on a fixed set of real-world programs as its primary measure of fuzzer effectiveness.

While standardized benchmarks are a significant step forward and in the right direction toward rigorous evaluation, these benchmarks are only as good as the underlying, assumed premise that the measurements are accurate and reliable. Previous work has discussed and investigated various aspects of benchmarking, like the choice of benchmarks [18], metrics [1], or statistical tests [8, 17], and issues with the underlying data (e.g., instrumentation flaws resulting in fuzzers not recording all coverage [10]), but the output of the evaluation tools is, generally, simply trusted and assumed to be correct.

Fuzzing Workshop (FUZZING) 2026
27 February 2026, San Diego, CA, USA
ISBN 978-1-970672-04-6
https://dx.doi.org/10.14722/fuzzing.2026.23030
www.ndss-symposium.org

In this paper, we challenge that assumption. On the basis of the FuzzBench evaluation pipeline, we investigate and aim to answer the following research questions:

**RQ 1:** Which sources of code coverage inconsistencies exist in the FuzzBench benchmark pipeline?

**RQ 2:** How large is the impact of each issue on the results?

**RQ 3:** Is the impact sufficient to affect how fuzzers should have been ranked in past evaluations? Could they call into question comparisons of prior work?

### A. Motivating Examples

There are many reasons why code coverage inconsistencies can occur. Here, we describe two motivating examples we encountered during previous FuzzBench experiments: a race condition in the coverage collection for SQLite and inaccurate coverage reports due to residual state in HarfBuzz.

*a) SQLite:* An examination of a SQLite coverage report revealed an anomaly: we observed `llvm-cov` reporting 18.4E $\approx 2^{64}$ executions of a single branch, which is evidently incorrect. Listing 1 shows an abbreviated version of the code. This issue occurred because multiple concurrent executions of the affected function raced to update the same instrumentation counters, occasionally missing an increment. This can be problematic because LLVM's profiling instrumentation computes the execution count of some branches based on the counters of others [15]. For example, it will compute how often the `else` branch of an `if` was taken by subtracting the number of times the condition was true from the number of times the condition was evaluated. An incorrect counter value due to a race condition caused this calculation to underflow, resulting in a wildly incorrect execution count. Both of these two motivating examples raise the question of how prevalent such issues are generally and they highlight the need for thorough evaluations of FuzzBench and other tooling that we rely on, but, so far, simply assumed to be correct.

*b) HarfBuzz:* For HarfBuzz, we observed an instance where we were unable to attribute the coverage of a specific branch to any single test case in our corpus. When we evaluated test cases individually, coverage was different from when we collected coverage for the entire test case corpus in the same process (which is how FuzzBench evaluates coverage). Our in-depth analysis revealed that this is due to residual state inside the target itself. Specifically, to test handling of low-memory conditions, HarfBuzz's testing setup uses an internal pseudorandom number generator (PRNG) to introduce allocation failures at random-but-deterministic locations. Each allocation advances the PRNG state and thus affects whether the next allocation triggers an artificial allocation failure. In essence, the sequence and order of test cases that were previously executed decides whether any specific test case (or the fuzzing campaign as a whole) can cover the corresponding error handling code.

### B. Contributions

In this paper, we perform the first systematic, in-depth analysis of the FuzzBench coverage measurement pipeline, to identify and categorize all such sources of inconsistencies. We identify several stages where errors can be introduced:

1) during snapshotting, due to incorrect corpus selection and failure to capture state in the target;
2) within the profiling instrumentation, through violating assumptions that cause miscomputations;
3) in the translation from raw counters to code coverage, due to mismatched function hashes, and finally;
4) in the way FuzzBench reports coverage, by summarizing template instantiations and miscounting branches.

Based on our analysis, we propose a set of new experiments to measure the impact of the discovered issues. We design our experiments specifically to investigate how much current FuzzBench evaluations may be affected and to inform future research on how to build more reliable evaluation frameworks.

In summary, we make the following contributions:

- We perform a qualitative analysis and identify multiple, specific problematic points in the FuzzBench coverage pipeline that can introduce incorrect coverage information and lead to wrong results.
- We introduce new experimental methodology to quantify the impact of the identified issues, including incorrect corpus selection, statefulness, instrumentation errors, and mismatches in reporting conventions.
- We propose a plan to thoroughly evaluate the impact of applying the necessary fixes, providing a clear path towards more accurate and reproducible fuzzer evaluations.

We will make our code and data publicly available at `softsec.link/fz26.cov`.

## II. Background

### A. Existing Benchmarks

The need for standardized evaluation approaches for fuzzers has primarily been addressed through two categories of benchmarks, those measuring bug discovery and those measuring code coverage. Initial efforts to create bug-based benchmarks focused on synthetic bugs, like LAVA-M [3]. Later ones, like Magma [7], reintroduced real bugs into newer versions of the previously buggy software to recreate the vulnerabilities. More recently, Zhang et al. [24] proposed a method to automatically undo security fixes in code, creating the REVBUGBENCH benchmark. Additionally, the vulnerable DARPA Cyber Grand Challenge (CGC) programs were ported to Linux [4] and Guido proposed to use them as a benchmark [6].

In parallel, researchers introduced coverage-based benchmarks, starting with the Fuzzer Test Suite [5], followed by FuzzBench [12] and UNIFUZZ [9]. Building on FuzzBench, Ounjai et al. introduced GreenBench, which tries to reduce the required power by shortening evaluation timeouts [14].

More recent work aims to make benchmark construction more principled. Wolff et al. [18] discuss the biases that benchmark properties can introduce to fuzzer evaluation results and find that properties, such as the initial seed set and the execution time of individual benchmark programs, have a significant impact on the evaluation's outcome. To mitigate the impact, they propose varying these properties to obtain a more comprehensive view of the performance of the benchmarked fuzzers. Similarly, to address the biases of existing benchmark suites, Miao et al. [13] proposed a benchmark that *generates* programs with diverse features, aiming to avoid bias toward shared code characteristics among the benchmark programs.

### B. On the State of Fuzzing Evaluations

In 2018, Klees et al. [8] studied then-recent fuzzing publications and discovered multiple common methodological issues in their evaluations. They transformed the identified issues into a set of recommendations for future evaluations. These recommendations include the choice of an appropriate baseline, the use of a diverse set of benchmark programs, and evaluation on a meaningful performance metric across multiple, sufficiently long trials to account for the inherent randomness of fuzzing. Unfortunately, a more recent survey by Schloegel et al. [17] found that these recommendations were often not followed. Specifically, they observed that almost a fourth of the surveyed papers failed to compare against an appropriate baseline or state-of-the-art fuzzer, and that few papers provide reliable measures of uncertainty and statistical significance to assess the robustness of their results. As a result, they further propose a new set of concrete guidelines, including the use of well-established evaluation benchmarks, with FuzzBench listed as the only example of such a benchmark [16, guideline A 2.1].

### C. Do Utilized Metrics Support the Claims?

Böhme, Szekeres, and Metzman [1] study the correlation and ranking agreement between coverage-based and bug-based benchmarks. They find that while both metrics are strongly correlated, they do not strongly agree on the ranking of different fuzzers. Thus, they conclude that both metrics are valuable measures of fuzzer effectiveness. Importantly, they argue for using "classical" coverage metrics over fuzzer-specific measures. FuzzBench already uses the default profiling instrumentation provided by Clang/LLVM to obtain coverage information. This makes it a natural choice for researchers who want to evaluate following these recommendations, which makes it important to investigate in detail.

### D. FuzzBench Internals

Figure 1 shows how FuzzBench collects coverage data. The fuzzer ingests seeds, generates test cases from them, and executes its instrumented copy of the target (**target$_{\text{fuzz}}$**) on these new inputs. Depending on its internal feedback metrics (e.g., new entries in the coverage map obtained from target$_{\text{fuzz}}$), the fuzzer retains some of these test cases and stores them in a corpus directory on disk, and it discards others. Some fuzzers, like AFL++ and LibAFL, also store additional metadata. In regular intervals, FuzzBench creates so-called "snapshots" of the fuzzing progress. To do so, it collects all files from the corpus directory that were created or modified since the last snapshot was taken. Before the fuzzer is started, FuzzBench also creates an initial snapshot (snapshot 0), which contains only the initial seeds. Each snapshot can then be evaluated independently of the fuzzer using a version of the target compiled with Clang's frontend coverage profiling instrumentation (**target$_{\text{cov}}$**). While fuzzer-specific instrumentation often has to balance performance and detail, this profiling instrumentation is designed to measure exact execution counts for every code region and branch.

To evaluate the coverage achieved by a fuzzer, FuzzBench extracts the snapshot archives, and executes target$_{\text{cov}}$ on the files contained therein. The same build of target$_{\text{cov}}$ is used for all evaluated fuzzers to make the results comparable. The coverage binary writes the resulting counter values alongside some metadata to disk in a simple binary format (`profraw`). FuzzBench then takes the individual `profraw` files, each containing coverage information for one snapshot, and merges them using `llvm-profdata` to obtain the cumulative coverage information up to that point in time. Using `llvm-cov` and the target$_{\text{cov}}$ binary, FuzzBench produces a JSON summary of the coverage achieved at each snapshot, from which it ultimately generates coverage-over-time reports.

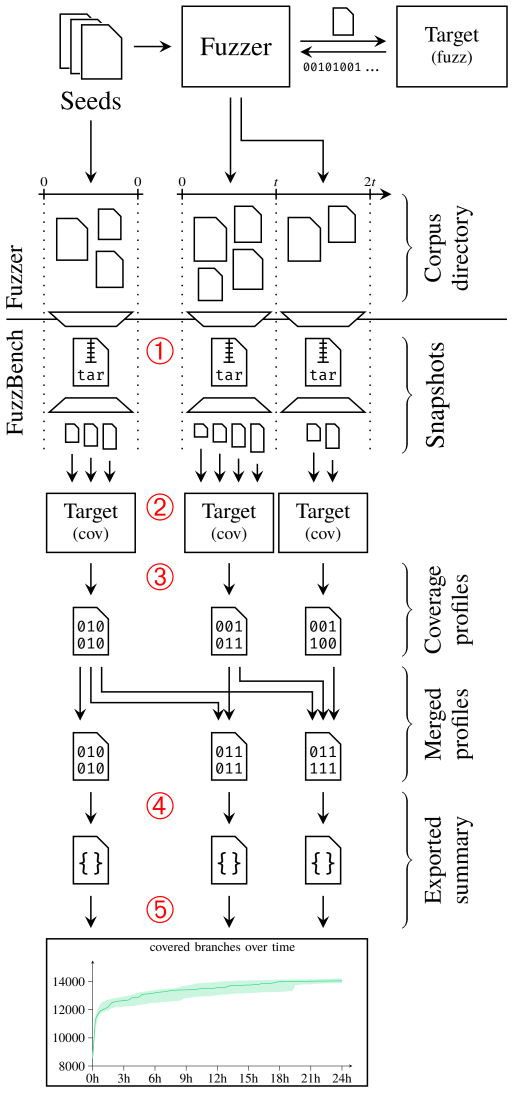


Figure 1: Overview of FuzzBench's coverage collection.

## III. Sources of Inaccuracies

In Section I-A, we gave two motivating examples of inaccuracies that we encountered while evaluating coverage reported by FuzzBench. We therefore want to systematically investigate all components of the coverage collection pipeline and look for causes of potential inconsistencies. The fuzzer and the benchmark program themselves are not the primary subjects of our evaluation because they can, in principle, be exchanged for any fuzzer/benchmark combination supported by the framework. Even though targets can behave non-deterministically, either of their own accord or due to interactions with the environment (e.g., system load or timeouts), this behavior is target-specific and cannot be comprehensively addressed from the benchmark suite's point of view. Therefore, we do not analyze any particular set of fuzzers or benchmarks but instead focus on the common parts and components of the FuzzBench infrastructure, as shown in Figure 1.

## A. ① *Snapshotting*

Coverage evaluations in FuzzBench start with snapshotting. In an ideal scenario, the snapshots capture the exact state of the fuzzing campaign at this time. However, upon closer inspection, we see that this stage can introduce several deviations from the original fuzzing campaign.

*a) Corpus Selection:* To realistically evaluate the coverage achieved by a fuzzer, $\text{target}_{\text{cov}}$ should be executed on exactly the same inputs as $\text{target}_{\text{fuzz}}$. We know that this is not the case, primarily because fuzzers do not store all generated test cases on disk for performance reasons.[1] Additionally, FuzzBench simply collects all files within its designated corpus folder, and some fuzzers store additional metadata and state files in this folder. For example, LibAFL-based fuzzers create lock files and JSON metadata, while those based on AFL++ store various metadata files in binary, CSV, and other plain text formats. In contrast, other fuzzers like honggfuzz and libFuzzer write additional files only to directories that will not be captured by the snapshotting, such as the current working directory or `/tmp`. This means that, for some fuzzers, $\text{target}_{\text{cov}}$ is executed on files that were never actually processed by $\text{target}_{\text{fuzz}}$, a problem that has been reported before in FuzzBench issue #1836 but remains unfixed.

*b) Statefulness:* Another issue, which we term *residual state*, may arise if the target carries over state between multiple executions of `LLVMFuzzerTestOneInput`. We have discussed an extreme example of such statefulness for HarfBuzz in Section I-A. But even without such explicit retention of state, there is almost always some amount of state that persists, including in subtle ways, such as in heap allocatior state after memory has been allocated or freed. Residual state might also differ depending on the execution order of the test cases or on whether and when state resets occur due to target restarts.

First, in FuzzBench, differences in residual state occur because it disregards the order in which the test cases were originally executed. The customized libFuzzer runtime that executes the test cases sorts them according to their size because it is used in "corpus merging" mode, where it prefers smaller test cases over larger ones that produce the same coverage. For corpus merging, this is sensible. However, it will almost always lead to an execution order different from the one in which the test cases were originally executed by the fuzzer, which means it results in different internal states.

Second, the evaluation process has no recollection of restarts that happened in $\text{target}_{\text{fuzz}}$ and has to insert artificial restarts at snapshot boundaries, which were likely not present during the original fuzzing campaign. In case of in-process or persistent mode forkserver fuzzers, $\text{target}_{\text{fuzz}}$ will execute multiple test cases before being restarted due to crashes, timeouts, or fuzzer specific criteria. Between individual restarts, $\text{target}_{\text{fuzz}}$ accumulates internal state, which affects coverage, and is reset whenever the target is restarted. These target restarts may have an impact on coverage that FuzzBench cannot reconstruct from the test case files on disk alone. For forkserver-based fuzzers that forcefully reset the target's state after each test case (e.g., some snapshot-based fuzzers), the same is true: While there was no opportunity for residual state in the original campaign, the coverage evaluation uses an in-process version of the target that retains residual state across executions.

Finally, because fuzzers do not usually record *all* test cases passed to $\text{target}_{\text{fuzz}}$, it is in general infeasible for FuzzBench to attempt to reconstruct the residual state in $\text{target}_{\text{cov}}$ exactly.

## B. ② *Profiling Instrumentation*

The next stage in the coverage evaluation pipeline runs the test cases through $\text{target}_{\text{cov}}$. While fuzzer instrumentations typically suffer from loss of accuracy, for example due to small counter bit widths or collisions in the coverage map, LLVM's profiling instrumentation is unaffected by such issues. It is used in situations that have a higher tolerance for performance degradation and, thus, it trades execution speed for additional precision. Prior work [20, 21, 22, 23] scrutinized profiling instrumentations generally. Instead of replicating them, we focus on issues arising specifically in fuzzing evaluations.

*a) Miscomputations:* To reduce both the performance overhead and the size of the generated profiles, the profiling instrumentation attempts to elide some counters by replacing them with mathematical expressions of other counter values. For example, it assumes that it can derive how often the `else` branch of an `if` statement was taken by subtracting the number of times the `then` branch was executed from the counter of how often the condition was evaluated. Under typical profiling workloads, these assumptions are perfectly sound: In a well-behaved program with well-behaved inputs, there is generally no way for control flow to leave the condition and not enter either of the `then` or `else` branches. However, there are no such guarantees for typical inputs and targets of fuzzing campaigns. Quite the contrary, fuzzers aim to uncover bugs in programs by generating new, pathological inputs and making the programs fail. In turn, we regularly violate these assumptions in coverage evaluations. For example, test cases that crashed the target or that were interrupted by a signal (e.g., due to exeeding the time limit for an individual input) also contribute coverage. This can then lead to branches being erroneously counted as covered. Additionally, there exist known issues for example when interacting with functions that can return multiple times (e.g. the `setjmp` and `longjmp` family of functions), which the instrumentation does not expect.[2] It is also known (and documented) that the default coverage instrumentation is vulnerable to races when used in multi-threaded programs [2]. The documentation explicitly warns that "the default [counter update mode] is `single`, which uses non-atomic increments. The counters can be inaccurate under thread contention." As we have shown (cf. Section I-A), this can lead to situations where branches are marked as covered, even though they were never executed. Users have also observed such miscomputations in OSS-Fuzz.[3]

[1]Arguably, this is also a function of how good the fuzzer is at identifying "interesting" test cases. Lipp et al. [10] investigate the impact of this pre-filtering by the fuzzer on code coverage metrics.

[2]For example, see LLVM issues #50119, #36480, #36473, and #36429.

[3]For example, see OSS-Fuzz issue #13483.

### C. ③ Retention of Counters

By default, LLVM's profiling instrumentation only writes the collected coverage data to disk when the program exists cleanly. As we established, this is not a sound assumption for fuzzing. To address this, FuzzBench uses a patched libFuzzer runtime that saves coverage data on crashes and timeouts. In contrast, other fuzzing frameworks, like OSS-Fuzz, use LLVM's default runtime for coverage collection and thus miss all coverage that has been recorded in a run prior to a crash.[4]

To avoid reexecuting all previous test cases for later snapshots, FuzzBench writes the coverage data of individual snapshots to disk and later merges the raw counters. This merging step is a simple addition of the raw counters from the `profraw` files and does not introduce further inconsistencies in practice. While the 64-bit wide counters could theoretically overflow, this would require a single code region to be executed $2^{64}$ times, which we consider impractical.

### D. ④ From Raw Counters to Code Coverage

The transformation from the raw counts to useful coverage information is based on the coverage mappings embedded into the coverage evaluation binary at compile time. These mappings include both the information which counter maps to which section of the source code, and the expressions we examined in Section III-B.

When generating coverage reports from the raw counts, LLVM attempts to ensure that the profile actually matches the coverage mapping embedded in the binary. Because there is no easy way to permanently link both (short of embedding the binary in the profile, which would be prohibitively space-inefficient), functions are instead assigned a simple structural hash at compile time. During evaluation, `llvm-cov` compares the hashes in both files and discards any coverage data where the hashes do not match. We have observed cases where function hashes in the profile do not match those in the binary.[5] In such cases, `llvm-cov` (and therefore also FuzzBench) is effectively blind to all coverage achieved by fuzzers in functions with mismatched hashes, and underreport their coverage.

There exists also (at least) one additional bug in which Clang generates an incomplete mapping between code and counters. At some point in the compilation pipeline, nested macro expansions inside of `while` loops lose their associated expansion regions. While the coverage data can be accurately computed from the expressions and remaining mappings, it is not included in the coverage summary and reports.[6]

### E. ⑤ From Coverage Data to FuzzBench Reports

Finally, FuzzBench ingests and interprets the data produced by `llvm-cov` to create a concise report tailored to fuzzing evaluations. However, this stage suffers from mismatched definitions and misunderstandings of the coverage output.

[4]For example, see OSS-Fuzz issue #13791.

[5]See also FuzzBench issue #1930 and LLVM issue #72786.

[6]See LLVM issue #176953.

*a) Summarization:* LLVM's coverage tooling summarizes the coverage of multiple function instantiations (e.g., from C++ templates) as the maximum number of executed and total branches across all instantiations. However, it obtains those values separately, that is, possibly from *different* instantiations.[7] In the context of fuzzing, this means that `llvm-cov` will underreport the number of covered branches and the total branch count. A fuzzer that covers all branches in all template instantiations is then considered equivalent to another that covers only the single largest instantiation. The HarfBuzz benchmark is an example where this summarization leads to a much lower branch count than counting each instantiation individually, as the project makes extensive use of templates. Since a function implementation may contain bugs in only a subset of its instantiations (e.g., depending on the template parameters), fuzzing all instantiations and achieving coverage in them is clearly more valuable than doing so for only one.

*b) Unique Branches:* FuzzBench further attempts to highlight differences between fuzzers by analyzing the coverage reports for branches that are uniquely covered by any of the fuzzers. For this analysis, it collects all covered branches for all functions from the report and calculates the size of the pairwise set differences between the individual fuzzers. However, FuzzBench *in this case only* uses a different definition of a branch. While LLVM counts the `true` and `false` cases of a conditional as two individual branches, FuzzBench considers the combination of the two as a single branch. Therefore, it fails to capture the difference in coverage between a fuzzer that only ever takes the `true` branch to another that only ever takes the `false` path. Notably, while the total count of covered branches used for coverage-over-time plots uses the summarization of all instantiation groups, the unique branch counting is done on each instantiation individually.

## IV. Experiment Design

Our qualitative analysis (cf. Section III) has shown that there are various places in the FuzzBench coverage evaluation pipeline where inaccuracies in the reported coverage can arise. For future evaluations, these methodological issues should be addressed regardless of their ultimate impact. However, it is currently unclear *how much* these issues actually affect the results of fuzzing evaluations. We aim to quantify their impact (**RQ 2**) and determine whether they might affect fuzzing evaluations of prior work (**RQ 3**). In the following, we discuss our proposed evaluation approach and preliminary results.

### A. General Setup

Following best practices [16], we obtain fuzzing results from FuzzBench by running each fuzzer for 24 hours on each of the benchmarks. We repeat this 10 times each to account for the randomness inherent to the fuzzing process.

For every trial, we evaluate the coverage in various scenarios (Sections IV-D to IV-I) to isolate the impact of each issue we identified in Section III. For our evaluation, we will repeat each experiment 10 times to control for non-determinism.

[7]We are not aware of the specific reasoning behind this design choice.

## B. *Target Selection*

We perform our evaluation on AFL++, LibAFL, and libFuzzer, which are the three fuzzers that most recent fuzzers are based on [11], as well as honggfuzz, which is frequently used in industry and academia. This allows a direct assessment of the most important baseline fuzzers in the ecosystem and ensures that our results can be generalized across fuzzers.

Our preliminary experiments are restricted to three of the 23 FuzzBench benchmarks (Bloaty, HarfBuzz, and SQLite). We will extend this to the full benchmark suite. We selected HarfBuzz and SQLite because they were the motivating examples (cf. Section I-A), and Bloaty because it is one of the largest FuzzBench targets in terms of lines of code.

## C. *Preliminary Evaluation Setup*

We conducted our preliminary experiments on two identical machines with 2,048 GiB of DDR5 RAM and two AMD EPYC 9754 CPUs with 128 physical cores running at 2.25 GHz each, that is, 256 physical cores total per server. We used Debian 12.18 and Docker version 29.0.4. To guarantee exclusive access to a dedicated physical core with a consistent CPU frequency to each fuzzer throughout our experiments, we disable hyperthreading (no logical cores), Turbo Mode, and dynamic frequency scaling.

FuzzBench pins each fuzzing instance to a dedicated physical core, preventing them from competing for CPU resources. Per server, we use 230 cores for fuzzing and 15 for coverage measurements, leaving 11 cores idle. This avoids performance degradation due to system or FuzzBench orchestration needs.

We use the benchmarks and fuzzers from revision `2a2ca6a` of FuzzBench. For Section IV-E, we extend the exact LLVM version used by that revision of FuzzBench (`bf7f8d6` with the appropriate FuzzIntrospector patches) to ensure that only our changes introduce differences to the baseline.

## D. ① *Snapshotting*

In Section III-A, we identified two possible issues in how FuzzBench collects and processes snapshots. We compare the coverage obtained by the baseline with the results when excluding non-testcase files from the snapshots. Generally, the format of metadata files is unlikely to be similar to the format parsed by the fuzzer target (in our preliminary evaluation, the metadata files did not contribute additional coverage), but this ultimately depends on the fuzzer and benchmark combination.

More significant is the impact of residual state in the target. We compare three distinct scenarios: First, we consider FuzzBench's default behavior (residual state between stored test cases). Then, we perform a fully stateless evaluation, in which the coverage binary is restarted for each input.

The final scenario is designed to match and reconstruct the original execution context as closely as possible. For this, we will modify the fuzzer to record *all* inputs to and log every restart of target$_{\text{fuzz}}$. We will account for the performance overhead of this modification by running the modified fuzzer not for a fixed amount of time, but for the same number of test case executions as the baseline, unmodified fuzzer.

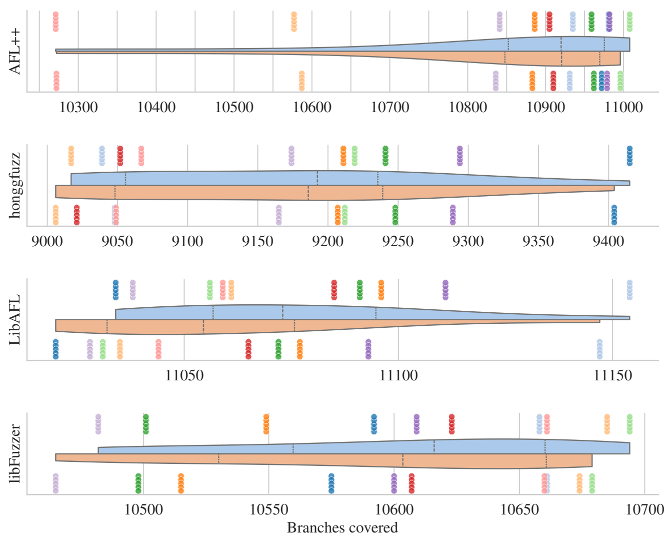


Figure 2: Branch coverage of different fuzzers on HarfBuzz with ● and without ● residual state. Individual evaluations of the same trial's corpus are shown as color-matched dots.

During the evaluation phase, we will supply the original test cases to target$_{\text{cov}}$ in the exact order in which they were supplied to target$_{\text{fuzz}}$. Setting aside any non-determinism in the benchmark itself, this reconstructs the original state of target$_{\text{fuzz}}$ in target$_{\text{cov}}$. To guarantee a fair comparison, we will only accumulate the coverage of those test cases that would also have been stored to disk by the unmodified fuzzer.

Our preliminary experiments compare the baseline to a stateless evaluation. Figure 2 shows the results of this experiment for HarfBuzz. It is clear that the same corpus processed with FuzzBench's default stateful mechanism (●) can report very different coverage than if the test cases are reevaluated without residual state (●). We suspect that the impact of restoring the "correct" state that originally made the test case relevant to the fuzzer will be even larger than the residual state introduced by FuzzBench's current evaluation.

## E. ② *Profiling Instrumentation*

To evaluate the impact of miscomputations in Clang's instrumentation due to race conditions, crashes, and the use of `setjmp` or `longjmp`, we propose comparing three different instrumentation methods.

As the baseline, we use Clang's default frontend instrumentation, as it is used in FuzzBench today. Then, we force LLVM to emit *atomic* counter increments instead, which specifically eliminates race conditions but not any of the other issues. Finally, we modify Clang to insert counters for all branches instead of deriving branch counts from other counters. This introduces some slight overhead, but eliminates the possibility of miscomputations entirely.

Unfortunately, while the SQLite example from Section I-A is easy to reproduce, the overall impact is masked by non-determinism in the target (it is reachable by design, e.g., via `SELECT random()`). For Bloaty, the only target in our

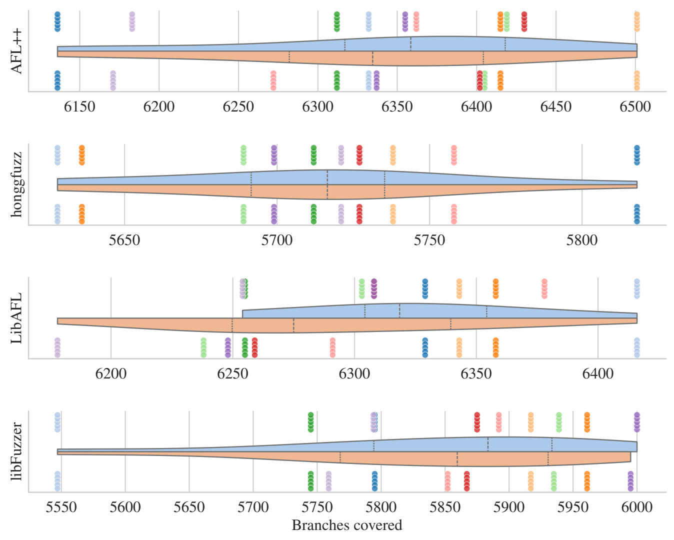


Figure 3: Branch coverage of different fuzzers on Bloaty with 🔵 and without 🟠 keeping coverage upon encountering a crashing input.

| Fuzzer | Bloaty | | HarfBuzz | |
|---|---|---|---|---|
| | Summary 76 184 total | All 83 580 total | Summary 17 228 total | All 32 658 total |
| AFL++ | 6 358.5 (8.35%) | 6 920.5 (8.28%) | 10 920 (63.39%) | 21 044 (64.44%) |
| honggfuzz | 5 716.5 (7.50%) | 6 219 (7.44%) | 9 192.5 (53.36%) | 17 652 (54.05%) |
| LibAFL | 6 318.5 (8.29%) | 6 892.5 (8.25%) | 11 073 (64.27%) | 21 042.5 (64.43%) |
| libFuzzer | 5 883.5 (7.72%) | 6 430 (7.69%) | 10 616 (61.62%) | 20 101.5 (61.55%) |

Table I: Evaluating all instantiations of a template individually instead of summarizing them results in different relative coverage for Bloaty and HarfBuzz. For HarfBuzz the effect even leads to a rank change between AFL++ and LibAFL. We report median results of 10 trials. The best performing fuzzer according to the metric in the column is highlighted.

preliminary experiments where fuzzers found crashes that might affect the coverage, the results show that the non-computing instrumentation consistently scores slightly lower than the default. This matches our expectations: It is easier for the default instrumentation to toggle a branch from zero to nonzero than vice-versa, so removing miscomputations should generally reduce the observed branch count. However, the differences we observed require additional validation and analysis to attribute them to a specific cause.

### *F. ③ Retention of Counters*

To reiterate the importance of correctly handling crashes and timeouts in the corpus, we compare the baseline results with those obtained after reverting the libFuzzer patch that FuzzBench uses to make its target$_{\text{cov}}$ resistant to crashes.

Figure 3 illustrates the results of our preliminary experiment on Bloaty. While honggfuzz (2 crashing inputs across 10 trials) is barely affected, LibAFL (409 crashing inputs across 10 trials) loses significantly more coverage.

### *G. ④ From Raw Counters to Code Coverage*

Our qualitative analysis (cf. Section III-D) revealed that hash mismatches can occur between the coverage evaluation binary and the coverage profiles within an evaluation campaign. The root cause of this is unknown; our research suggests that function entries with the correct hash are generated during compilation but discarded, and that retaining them allows evaluating the coverage obtained on these functions.

Systematically evaluating the impact will likely require an additional patch to Clang or LLVM, but manual investigation of the `profdata` files from our preliminary experiments on Bloaty shows that these functions are indeed covered (so FuzzBench underreports the branch coverage), but by all fuzzers on all branches, that is, it affects all fuzzers equally.

In our experiments, we want to further investigate the root cause of this issue and also quantify its impact by comparing the current instrumentation (baseline) to one compiled with the fix in place (no hash mismatches).

Similarly, we want to further investigate and ultimately fix the root cause behind the missing expansion regions identified earlier. Given a fix, we will perform a similar evaluation to quantify the impact of this issue.

### *H. ⑤ From Coverage Data to FuzzBench Reports*

The total coverage values reported by `llvm-cov` summarize all instantiations of templated functions. We want to quantify how many of these individual instantiations are actually covered by fuzzers. To do so, we directly evaluate the branches of each function, including all instantiations, and compare this detailed count to the default summarized coverage.

Our preliminary results indicate that, for the C++ projects Bloaty and HarfBuzz, there is a significant gap between the number of branches available for evaluation and those reported by the summary. When examining instantiations individually, HarfBuzz reports almost 90% more branches, and the top-ranked fuzzer (by the median of all 10 trials) switches from LibAFL to AFL++. Table I shows these results in more detail.

Additionally, FuzzBench's notion of a "unique" branch disagrees with LLVM's definition of covered branches. Therefore, we will compare the unique branch count as currently reported by FuzzBench to an adapted version that considers both `true` and `false` paths of a conditional branch individually (two-sided unique branches). Figure 4 shows the results across all 10 trials. Again, we observe significant differences between both variants. Particularly noteworthy is the set of branches in HarfBuzz that honggfuzz covered but libFuzzer did not: FuzzBench would report no improvement at all, but the raw coverage data reveals that there are 12 additional branches.

### *I. Overall Impact*

Finally, after evaluating each issue in isolation, we will compare the overall impact of all fixes together against a plain baseline run. This way, we can assess the combined impact of all discovered issues. We will also compare these results to those obtained using a new Clang version, which incorporates patches for other bugs fixed in the meantime.

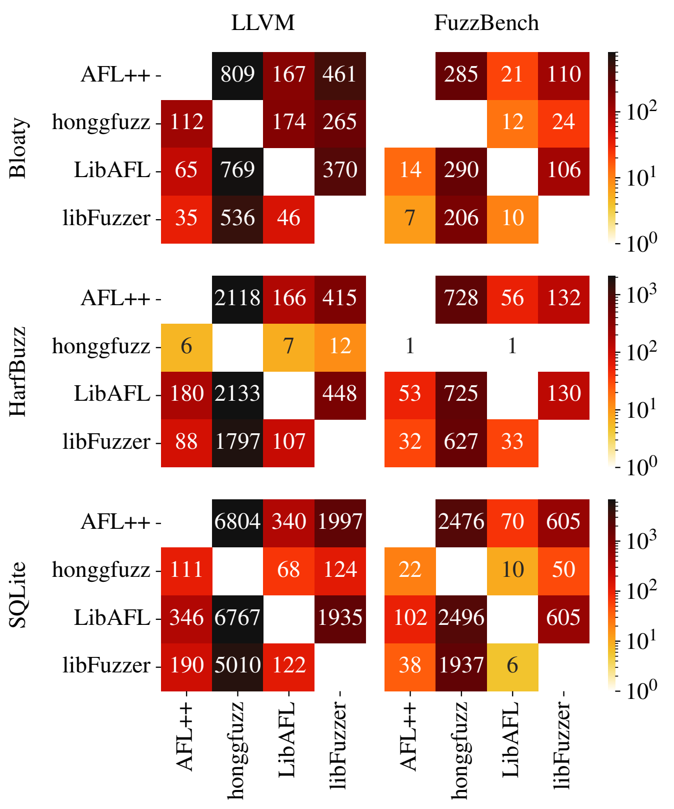


Figure 4: Pair-wise unique branch coverage as counted by FuzzBench and the branch definition of LLVM. The fuzzer in the row covered $n$ branches that the one in the column did not.

To assess the real-world impact of our findings (**RQ 3**), we will conduct a review of existing evaluations. With data from literature, we can identify cases where the inconsistencies introduced by the coverage evaluation could have affected the ranking of the individual fuzzers, and which evaluations have a margin that is sufficiently large to not be affected.

## V. Related Work

*a) Measured Coverage vs. Actual Coverage:* Current code coverage evaluations are performed on the test cases a fuzzer has kept for further mutation or deemed otherwise interesting, for instance, because it crashed the target. However, Lipp et al. [10] have shown in a small study that the shortcomings of AFL's coverage instrumentation lead it to discard inputs that account for up to 9% of the basic blocks covered during a campaign. This means that coverage evaluations based on the fuzzer's corpus underestimated the actually achieved coverage in their experiment. Our work focuses on a different source of error, namely the utilized coverage collection framework.

*b) Scrutinizing Coverage:* Previously, members of the software engineering community have scrutinized the correctness of code coverage tooling. Yang et al. [20] proposed metamorphic testing to find bugs in `llvm-cov` and `gcov` by mutating programs and utilizing metamorphic relations between the coverage results of the original and the mutated version. In a different work, they proposed to generate random programs using CSmith [19] and then use differential testing to find disagreements between different coverage tools [22]. Decov [21] is another approach that does not rely on the output of the code coverage tooling but instead cross-validates it with debugging data. All prior work tests either on synthetic or deterministic programs from the GCC compiler test suite. While this proved successful in discovering bugs in the coverage instrumentation and tooling itself, the setup is very different from FuzzBench, where we collect coverage on large, complex, real-world programs with many different inputs.

DebCovDiff [23] closes the gap between simple programs and real-world software by testing the coverage collection on Debian packages. They inspect differences between `llvm-cov` and `gcov` to find bugs in them. While these tests are successful for the purpose of uncovering discrepancies between the coverage tools, they cannot uncover issues with the evaluations of fuzzing campaigns, as they do not consider circumstances like crashes, timeouts and a large number of inputs that make triggering a data race more likely.

## VI. Looking Ahead

In this registered report, we present the first comprehensive analysis of the FuzzBench coverage collection pipeline and propose an extensive evaluation plan to quantify the impact of the issues we uncovered during our analysis.

Our investigation has shown that there is a clear need for improvement in the current FuzzBench methodology. Nevertheless, we strongly believe that standardized benchmarks are the right direction for the fuzzing community. Through scrutiny of such benchmarks, like the one we proposed in this work, we can ultimately increase confidence in all research and projects that use and depend on them for their evaluations.

Despite their flaws, we do believe that FuzzBench and `llvm-cov` remain the right approach and tools for the job. Because their coverage collection technique is entirely agnostic to the instrumentation used by the fuzzers themselves, this independence guarantees that the evaluation metric does not provide an unfair advantage to any particular fuzzer that might share an instrumentation heritage with the measurement tool.

We will make our code, and artifacts, publicly available at `softsec.link/fz26.cov`. We commit to open science, and hope our findings will help improve FuzzBench and contribute to a more robust foundation for future fuzzing research.

## Acknowledgments

We thank our anonymous reviewers for their suggestions on further improving the presentation and experiment design.

This research is supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA – 390781972, as well as the Vienna Science and Technology Fund (WWTF) and the City of Vienna [Grant ID: 10.47379/ICT19056]. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the respective funding agencies.

## References

[1] M. Böhme, L. Szekeres, and J. Metzman. “On the reliability of coverage-based fuzzer benchmarking.” In: *Proceedings of the 2022 44th IEEE/ACM International Conference on Software Engineering (ICSE)*. May 2022. DOI: 10.1145/3510003.3510230.

[2] *Clang Compiler User’s Manual — Clang 15.0.0 documentation*. URL: https://releases.llvm.org/15.0.0/tools/clang/docs/UsersManual.html (visited on 12/10/2025).

[3] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. “LAVA: Large-Scale Automated Vulnerability Addition.” In: *Proceedings of the 37th IEEE Symposium on Security & Privacy (S&P)*. May 2016. DOI: 10.1109/sp.2016.15.

[4] K. El-Faramawi, L. Maggiore, A. Dinaburg, P. Goodman, R. Stortz, and J. Little. *DARPA Challenge Binaries on Linux, OS X, and Windows*. Version 810d7b2. Trail of Bits, Dec. 27, 2022. URL: https://github.com/trailofbits/cb-multios (visited on 12/04/2025).

[5] Google. *Fuzzer Test Suite*. 2017. URL: https://github.com/google/fuzzer-test-suite (visited on 10/09/2025).

[6] D. Guido. *Your tool works better than mine? Prove it.* Trail of Bits. Aug. 1, 2016. URL: https://blog.trailofbits.com/2016/08/01/your-tool-works-better-than-mine-prove-it/ (visited on 12/04/2025).

[7] A. Hazimeh, A. Herrera, and M. Payer. “Magma: A Ground-Truth Fuzzing Benchmark.” In: 4.3 (Dec. 2020), pp. 1–29. DOI: 10.1145/3428334.

[8] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. “Evaluating Fuzz Testing.” In: *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Oct. 2018. DOI: 10.1145/3243734.3243804.

[9] Y. Li, S. Ji, Y. Chen, S. Liang, W. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng, K. Lu, and T. Wang. “UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers.” In: *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*. Aug. 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/li-yuwei.

[10] S. Lipp, D. Elsner, T. Hutzelmann, S. Banescu, A. Pretschner, and M. Böhme. “FuzzTastic: a fine-grained, fuzzer-agnostic coverage analyzer.” In: *Proceedings of the 2022 44th IEEE/ACM International Conference on Software Engineering (ICSE)*. May 2022. DOI: 10.1145/3510454.3516847.

[11] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. “The Art, Science, and Engineering of Fuzzing: A Survey.” In: *IEEE Transactions on Software Engineering* 47.11 (Nov. 2021). Data as published in revision 4f99e94 (2025-09-13) of https://github.com/SoftSec-KAIST/Fuzzing-Survey/, pp. 2312–2331. DOI: 10.1109/TSE.2019.2946563. (Visited on 12/10/2025).

[12] 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 Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. Aug. 2021. DOI: 10.1145/3468264.3473932.

[13] M. Miao, S. Kummita, E. Bodden, and S. Wei. “Program Feature-Based Benchmarking for Fuzz Testing.” In: (June 2025), pp. 527–549. DOI: 10.1145/3728899.

[14] J. Ounjai, V. Wüstholz, and M. Christakis. “Green Fuzzer Benchmarking.” In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. July 2023. DOI: 10.1145/3597926.3598144.

[15] A. Phipps. “Branch Coverage: Squeezing more out of LLVM Source-based Code Coverage.” 2020 LLVM Developers’ Meeting. Sept. 2020. URL: https://llvm.org/devmtg/2020-09/slides/PhippsAlan_BranchCoverage_LLVM_Conf_Talk_final.pdf (visited on 12/11/2025).

[16] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz. *Fuzzing Evaluation Guidelines*. Version 1.0.3 (29a8f5c). Mar. 7, 2024. URL: https://github.com/fuzz-evaluator/guidelines (visited on 12/05/2025).

[17] 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: *Proceedings of the 45th IEEE Symposium on Security & Privacy (S&P)*. May 2024. DOI: 10.1109/sp54263.2024.00137.

[18] D. Wolff, M. Böhme, and A. Roychoudhury. “Fuzzing: On Benchmarking Outcome as a Function of Benchmark Properties.” In: *ACM Transactions on Software Engineering and Methodology* 34 (Apr. 2025). Just Accepted. DOI: 10.1145/3732936.

[19] X. Yang, Y. Chen, E. Eide, and J. Regehr. “Finding and understanding bugs in C compilers.” In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2011. DOI: 10.1145/1993498.1993532.

[20] Y. Yang, Y. Jiang, Z. Zuo, Y. Wang, H. Sun, H. Lu, Y. Zhou, and B. Xu. “Automatic Self-Validation for Code Coverage Profilers.” In: *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Nov. 2019. DOI: 10.1109/ASE.2019.00018.

[21] Y. Yang, M. Sun, Y. Wang, Q. Li, M. Wen, and Y. Zhou. “Heterogeneous Testing for Coverage Profilers Empowered with Debugging Support.” In: *Proceedings of the 31st Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. Nov. 2023. DOI: 10.1145/3611643.3616340.

[22] Y. Yang, Y. Zhou, H. Sun, Z. Su, Z. Zuo, L. Xu, and B. Xu. “Hunting for bugs in code coverage tools via randomized differential testing.” In: *Proceedings of the*

*41st IEEE/ACM International Conference on Software Engineering (ICSE)*. May 2019. DOI: 10.1109/ICSE.2019.00061.

[23] W. Zhang, J. Jia, E. Yu, D. Marinov, and T. Xu. “DebCovDiff: Differential Testing of Coverage Measurement Tools on Real-World Projects.” In: *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Nov. 2025. DOI: 10.1109/ASE63991.2025.00094.

[24] Z. Zhang, Z. Patterson, M. Hicks, and S. Wei. “FIXREVERTER: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing.” In: *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*. Aug. 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/zhang-zenong.

## Appendix A
## Revision Requirements

The reviewers encourage the authors to address the following points:

1) Complete the proposed evaluation and repeat fuzzing campaigns for at least 30 trials as recommended by Klees et al. [8]. Report results using meaningful aggregate statistics (e.g., mean and standard deviation), and include appropriate statistical analyses (such as confidence intervals and hypothesis testing) to better understand the effects introduced by the identified issues.
2) In addition to `llvm-cov`, the authors should include comparisons with `gcov` where feasible to help distinguish tool-specific artifacts and strengthen the evaluation.
3) For issues such as the counter-to-coverage translation discussed in Section IV-G the paper should either identify the root cause or clearly outline risk-mitigation strategies if a definitive fix cannot be found. The final version should avoid leaving major issues inconclusive without discussing such mitigations.
4) Make the answer to **RQ 1** explicit. Reviewers also suggest summarizing all identified sources of coverage inconsistency in a table, ideally including references to supporting evidence (e.g., bug reports, issues, or experiments) and pointers to how each source is evaluated.
5) Evaluation methodology and setup should be more concise, and include whether fuzzing is performed in parallel or on the same machine, whether Docker is used, and how AFL is configured.
6) The threats to validity section should discuss generalisability and transferability for other benchmarking frameworks.
7) Revise Figure 2 for clarity and precision, including clearer labeling, captions, and alignment with the accompanying text.
8) Revisit the coverage evaluation pipeline to assess whether there are additional sources of inconsistency beyond those already identified, and discuss any newly discovered issues, if any.