SoK: A Systematic Review of Integration and Reproducibility of Fuzzing Research into AFL++

Nico Schiller[1], Nils Bars[1], Moritz Schloegel[2], Thorsten Holz[1]
[1]*Max Planck Institute for Security and Privacy, firstname.lastname@mpi-sp.org*
[2]*CISPA Helmholtz Center for Information Security, schloegel@cispa.de*

Abstract*—Fuzzing has become one of the most effective automated bug discovery techniques. Despite extensive research covering all aspects, it is difficult to assess the actual progress fuzzing has made over the years. In this paper, we present a large-scale empirical analysis of fuzzing progress using AFL++, the state-of-the-art fuzzer that continuously integrates research-driven improvements. Using 645,000 CPU-hours of experiments, we comprehensively measure how fuzzing has improved in terms of code coverage and evaluate the impact of various features over the years. Surprisingly, we find a plateau in exploring new program behavior: while some techniques yield isolated performance gains, overall progress in exercising new coverage has largely stalled. Studying whether our observations generalize to LIBAFL and FUZZILLI, we find our observations hold across all three fuzzers. To better understand this stagnation, we complement our empirical study with a survey of 405 peer-reviewed fuzzing papers published between 2018 and 2024 at the leading security and software engineering venues. We identify 60 papers that extend AFL/AFL++ and study how feasible the integration into the baseline is, and if the baseline fuzzer adopted it. Surprisingly, we observe little adoption in practice, with irreproducible results, reliance on complex external dependencies, and limited practical benefit as the main barriers. Discussing our analysis results with the AFL++ maintainers, we find a growing disconnect between academic research and real-world adoption, underscoring the need for stronger reproducibility standards and a more realistic benchmarking of proposed improvements.

1. Introduction

Scientific progress is based on the continuous refinement of existing knowledge. New methods build on previous discoveries and, in turn, enable more advanced approaches. Regular and rigorous assessment of scientific progress is crucial for verifying genuine advances, distinguishing innovations that yield substantial improvements from those that introduce only incremental refinements [1]–[4]. This principle applies broadly to all areas of (computer) science in general and to computer security in particular [5].

In software testing, rigorous empirical evaluation is crucial for identifying the most promising ideas in this rapidly evolving field. In recent years, fuzzing has emerged as one of the most effective testing techniques for finding software vulnerabilities. By automatically generating (semi-)random inputs to explore execution paths, fuzzers have uncovered thousands of critical bugs in widely used systems [6]–[8]. The impact of fuzzing is evident in both academia and industry, with hundreds of research papers, adoption by companies such as Google and Microsoft, and large-scale initiatives like OSS-Fuzz [8] discovering tens of thousands of real-world vulnerabilities.

Research on fuzzing is extensive and continues to grow [6], [7], [9], with proposed improvements ranging from smarter instrumentation techniques and novel input mutation strategies to enhanced scheduling mechanisms. Despite this diversity, all approaches share the common goal of effectively and reliably identifying software faults. But have these innovations meaningfully improved fuzzing effectiveness, or was the last major leap the introduction of coverage-guided fuzzing with AFL [10] in 2013? Answering this question is challenging and requires a comprehensive empirical evaluation.

In this paper, we systematically examine advances in fuzzing research over the past seven years. We conduct a large-scale empirical study to evaluate the evolution of AFL++, the state-of-the-art general-purpose fuzzer, and extensively review the *impact* of various improvements over time. To complement the picture and to draw an overarching conclusion for the field of fuzzing, we additionally tested two other widely used fuzzers: LIBAFL as an alternative to AFL++ and FUZZILLI, the de facto standard for fuzzing JavaScript engines in industry. To conduct this study, a consistent metric is essential for measuring progress. A standard and widely adopted indicator of a fuzzer's effectiveness is the *code coverage* it achieves. Coverage is particularly relevant for three reasons: First, a fuzzer cannot detect faults in unexplored code, and higher coverage correlates with a higher probability of finding bugs [11]. Second, coverage is straightforward to measure and has become a standard metric in fuzzing research: a recent study found that 77% of current fuzzing papers report some form of code coverage when evaluating their tools [9]. Finally, established evaluation guidelines for fuzzing recommend using code coverage (in the form of unique edges or executed basic blocks) as a consistent secondary metric alongside direct vulnerability-finding results [9], [12], [13]. Thus, we use coverage as our primary metric. We also conducted an experiment on bug-finding capability, but found it too coarse-grained to meaningfully measure progress across different versions of the same fuzzer. With this context in mind, we study the following research question in this paper:

> *How has fuzzing advanced in the past years, and can fuzzers now achieve substantially greater coverage?*

To answer this question, we systematically evaluate the

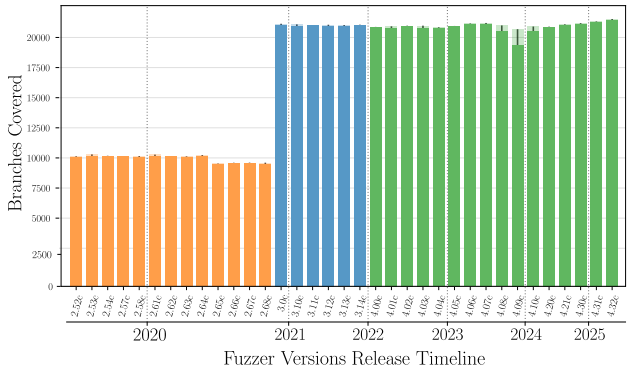


Figure 1: All AFL++ versions with default settings on the target SQLITE3 aligned with the respective release year. We use different colors to visualize major versions.

chronological development of AFL++ [14], a widely used general-purpose fuzzer that serves as a natural benchmark for assessing progress in fuzzing. As an evolution of the original AFL fuzzer [10], AFL++ was explicitly designed to integrate "incremental steps of fuzzing research" [14], and it is known for integrating numerous techniques proposed in academic research. In fact, many academic fuzzing tools are extensions of AFL++ (e.g., [15]–[21]). In addition, AFL++ (and, in older works, its predecessor AFL) is frequently used as a reference point for evaluating newly proposed fuzzing methods. If fuzzing research has advanced the state of the art, it should manifest in the evolution of AFL++. Studying its development thus offers a proxy for tracking the trajectory of fuzzing research itself.

Instead of testing every release, we carefully selected 10 representative versions of AFL++ that cover important milestones (e.g., major version changes or important features). To ensure that our results are meaningful and generalizable, we tested these versions across 15 real-world software projects spanning diverse application areas and input types. A key challenge in the long-term evaluation of fuzzers is the evolution of instrumentation and compiler behavior. To isolate the effects of AFL++ itself, we tested each version with both the latest compiler and the one available at the time of release. To ensure a fair and consistent comparison, we conducted all experiments in a controlled, containerized environment on identical hardware and repeated each ten times.

Based on more than 645,000 CPU hours (73.6 CPU years) of experiments, we carefully analyzed how different configurations and features influence AFL++'s ability to explore code. Our results indicate that key features, such as `CmpLog` and `dict2file`, significantly improved performance. However, these features were added years ago, and we find that improvements have largely stalled since then. An exemplary visualization for the target SQLITE3 is shown in Figure 1: after the significant increase in early 2021, progress has largely stalled. Through similar experiments with LIBAFL and FUZZILLI, we found that their code-coverage performance has also plateaued, consistent with our observations for AFL++.

To understand potential reasons *why* progress stagnated, we conducted a literature analysis of all 405 fuzzing papers published between 2018 and 2024 at the leading computer security and software engineering venues. Of these, 60 were based on AFL or AFL++, and 44 provided publicly available source code. We evaluated these papers based on their feasibility of integration into the respective baselines and examined which ones were actually adopted. Combining these insights with feedback we have received through discussion with the AFL++ maintainers, our analysis shows that only a small fraction of techniques have been integrated into AFL++. We identify several reasons for this, including overly complex implementations, non-reproducible results, and improvements that lack statistical significance. These factors hinder an integration into AFL++. Based on this meta-study, we distill three key lessons for the community: *To drive meaningful progress, research must prioritize not only novelty, but also integrability, reproducibility, and practical relevance.*

While integrability into a widely used baseline such as AFL++ is not a prerequisite for demonstrating the value of a research idea, our findings indicate that it strongly correlates with real-world impact. Techniques that can be integrated with low engineering overhead are significantly more likely to be adopted and maintained in practice. We therefore treat integrability not as a requirement but as a pragmatic lens for assessing whether research contributions translate into sustained usage.

Contributions. In summary, our key contributions are:

- We design a systematic and comprehensive evaluation of AFL++, one of the most widely used general-purpose fuzzers, which integrates successful fuzzing works. We examine how it has evolved in the last seven years in its instrumentation, mutation strategies, and scheduling mechanisms.
- We conduct an extensive empirical study spanning more than 645,000 CPU hours across multiple targets. Our core insight is that the effectiveness of AFL++ and other fuzzers has largely plateaued, despite many improvements in the scientific literature.
- We conduct a meta-study of the fuzzing literature by analyzing 405 peer-reviewed papers and systematically assessing their integration into AFL++. We find that only a small fraction of the proposed techniques are adopted, often due to limited generalizability, marginal practical benefits, or integration complexity.

We make our research artifacts accessible at https://doi.org/10.5281/zenodo.19887428.

2. Evolution of AFL++

Fuzzing remains an active research area, with substantial time and resources invested by academia and industry. One of the most widely used modern fuzzers is AFL++ [14], the successor to AFL [10]. Now the de facto standard for open-source fuzzing, AFL++ is frequently used as a baseline in evaluations of general-purpose fuzzers. It builds on AFL's coverage-guided approach and benefits from community-driven development. When new techniques or enhancements from academic research show improvements over the current AFL++ version, those features are often added to AFL++ to keep it up to date [22]–[24]. Since AFL++'s development began in 2019, numerous features have been incorporated, including improved instrumentation support, more effective scheduling strategies, and various mutation techniques. We now outline the

development history of AFL++, highlighting key features and changes to default settings.

2.1. Fuzzing Features of AFL++

Different settings, schedulers, and mutations can be used to fine-tune various aspects of the fuzzer, potentially influencing the fuzzing process. Some settings are set by default, but the defaults might change across versions. The following features directly impact the fuzzing process and can be controlled by a user.

Deterministic vs Havoc. AFL++ features different categories of mutations applied to test cases during the fuzzing process, broadly divided into deterministic and non-deterministic stages. In the deterministic stage, the fuzzer systematically applies predefined mutations, such as bit flips, byte swaps, arithmetic modifications, and inserting special values, to generate new test cases. These controlled modifications ensure that common edge cases are explored exhaustively. In contrast, the havoc stage consecutively applies multiple mutations to the same test case, including byte insertions, deletions, block replacements, and splicing from other inputs. These more advanced mutations allow the fuzzer to explore complex, non-trivial execution paths that deterministic methods may miss. Finally, the splicing stage merges two inputs and applies the havoc stage to the merged one.

Over the years, the default settings for these stages have changed, e.g., to incorporate updated mutators. The effectiveness of deterministic and havoc mutations can vary significantly. Deterministic fuzzing often takes days, as AFL++ applies all mutations exhaustively to every input. In contrast, the havoc stage enables faster exploration and typically reaches coverage saturation within 24 hours. As a result, the deterministic stage is often skipped in research, where experiments aim to maximize coverage within limited time frames. In version *4.10c*, the deterministic stage was completely rewritten [25], making direct comparisons with the original challenging. For readability, we refer to both variants simply as the deterministic stage.

Power Schedules. In AFL++, power schedules determine how much effort ("energy") the fuzzer allocates to each seed input to derive new test cases from this seed. Different algorithms are used to compute this energy, combined with the test case's performance score. Two notable schedules are `Explore` and `Fast`, proposed initially by Böhme et al. [22] and later incorporated first into AFL, then into AFL++. In the initial release of AFL++ (*2.52c*), the default schedule was `Explore`, which assigns a low but constant energy level to all seeds, ensuring balanced exploration across the entire input space. In contrast, the `Fast` schedule assigns energy inversely proportional to the path frequency exercised by the seed: seeds that exercise less-frequently executed paths receive higher energy, meaning more mutations are generated from them. This approach biases the fuzzer toward exploring less frequently executed paths, increasing the likelihood of discovering new program behaviors [14].

Over time, the default schedule setting has evolved based on newer experimental insights and bug fixes. Starting with version *3.0c*, the default schedule was changed from `Explore` to `Fast`. However, this change was later reverted in version *4.10c*. Users can manually select the desired power schedule via a command-line parameter.

CmpLog. `CmpLog` is an instrumentation and fuzzing feature added in AFL++ *2.61c* that enhances the fuzzer's ability to handle complex input comparisons by logging the operands involved in these comparisons into shared memory. This mechanism draws inspiration from REDQUEEN [24], which focuses on input-to-state transformations to bypass hardcoded magic bytes and similar obstacles in target binaries. By leveraging the logged comparison operands, AFL++ can perform targeted mutations, e.g., by inserting them into the input for the target. To use `CmpLog`, two versions of the target program must be built: one with standard AFL++ instrumentation and another with `CmpLog` instrumentation enabled [14], [26].

Dict2File. The `dict2file` feature was added in AFL++ *3.0c* to automate the extraction of useful mutation tokens. It uses an LLVM pass that extracts constant string comparison parameters from the target program's source code. This automatically generated dictionary file can then be passed at runtime, allowing the fuzzer to incorporate these values during mutation [26].

MOpt. `mOpt`, short for "Optimized Mutation Scheduling", is an enhancement integrated into AFL++ that aims to improve fuzzing efficiency by optimizing the selection probabilities of mutation operators. Originally proposed by Lyu et al. [23], `mOpt` leverages a customized particle swarm optimization algorithm to adjust the mutation strategies dynamically based on their effectiveness during the fuzzing process. In AFL++, `mOpt` is divided into two stages: a *Pilot* stage, which evaluates the operators and assigns probabilities based on the effectiveness, and a *Code* stage, which generates the mutations based on the probabilities from the Pilot stage. The first version of AFL++ that supports `mOpt` is version *2.53c*.

2.2. Instrumentation Features of AFL++

Modern coverage-guided fuzzers typically rely on source-based instrumentation via LLVM passes that add coverage information, enabling the fuzzer to track execution flow. AFL++ features different LLVM passes and, therefore, instrumentation options. These options can support the fuzzing process by splitting complex constraints, providing additional feedback, or reducing the likelihood of collisions. Besides that, AFL++ also supports the so-called *persistent mode*, which reduces the number of forks of the target binary and results in a considerable performance increase [14], [27]. For when no source code is available, AFL++ features a binary-only QEMU mode and a snapshot-based mode based on NYX [26], [28].

The available instrumentation features depend on the versions of AFL++ and `Clang`.

CompCov. The `CompCov` or `LAF-Intel` feature [29] has been available since the initial release of AFL++, utilizing LLVM passes. It enhances the fuzzer's ability to handle complex conditional statements within the target program by transforming complex comparisons into simpler ones during compilation. For instance, multi-byte comparisons are split into several single-byte comparisons, allowing the fuzzer to solve them consecutively [14].

AFL Classic Instrumentation. The `Classic` instrumentation was inherited from the original AFL, and is a lightweight coverage mechanism that instruments programs by assigning pseudo-random IDs to basic blocks at compile time. During execution, the technique computes an *edge ID* by XOR-ing the current and previously executed basic block IDs, which is then used as an index to update the coverage bitmap. While this approach offers minimal runtime overhead and simplicity, using an index based on pseudo-random IDs is prone to collisions, especially for large targets with many basic blocks [30]. These collisions may cause novel coverage to be missed.

CTX. *Context Sensitive Branch Coverage* (`CTX`) is an advanced instrumentation technique that extends traditional edge coverage of the `Classic` instrumentation by incorporating the calling context of each function: the coverage information for a function is distinguished based on its caller, allowing the fuzzer to differentiate between identical functions invoked from different locations [26].

LLVM-Native `PCGuard` and AFL++ `PCGuard`. AFL++ inherits the more effective `trace-pc-guard` instrumentation based on LLVM *SanitizerCoverage* [31] from the original AFL. This instrumentation is more precise than the `Classic` AFL approach by splitting critical edges and adding new instrumented dummy blocks. However, this approach still uses random IDs and hence collisions can occur. In AFL++ version *2.66c*, a collision-free `PCGuard` implementation based on LLVM's was added, replacing random IDs with a global counter.

3. Experimental Design and Setup

We design a comprehensive set of experiments to systematically evaluate the evolution of AFL++. To obtain meaningful results at an acceptable computation cost, we need to answer the following questions:

1) Which AFL++ versions should be evaluated?
2) Which targets should be selected?
3) Which features should be analyzed?
4) Which instrumentation modes should be analyzed?
5) How should the target be instrumented?
6) What kind of baseline configuration should be used?

General Setup. Following the recommendations of Klees et al. [12] and Schloegel et al. [9], we repeated each experiment ten times. All experiments were executed in a dockerized environment and conducted on the same hardware configuration: two AMD EPYC 9654 CPUs (192 physical cores / 384 logical cores) with 768GB of RAM and SSD storage as the backing medium. Each fuzzer is run in a dedicated container with one CPU assigned. To calculate code coverage, we compiled a coverage-instrumented binary of each target using LLVM 18 and replayed the generated fuzzing corpus.

AFL++ Version Selection. Due to the large number of features and the diversity of target types, testing all 34 released versions is computationally infeasible. We conducted a pilot experiment on two representative targets (SQLITE3 for structured inputs and BLOATY for binary input), using each version in its default configuration. We exemplarily depict the results for SQLITE3 in Figure 1 on page 2; the results for BLOATY are shown in Appendix C. We observed that several versions exhibited negligible differences, allowing us to safely exclude them from further evaluation. We further prioritized releases that introduced key features (e.g., `CmpLog` in *2.61c* or the revised deterministic mode in *4.10c*), and ensured broad `LLVM` version support. As a result, we selected ten AFL++ versions from *2.52c* to *4.32c*, including the first and last releases of each major version (e.g., *3.0c* and *3.14c*) to capture both incremental improvements and major changes. Table 1 summarizes our ten selected AFL++ versions, their evolving default configurations (e.g., deterministic mode, seed scheduling), and the availability of key features like `mOpt`, `CmpLog`, and `dict2file`.

TABLE 1: Evaluated AFL++ versions with the default settings for the scheduler and the deterministic mode (✓ use deterministic, ✘ skip-deterministic) as well as whether `CmpLog`, `mOpt`, and `dict2file` are available.

AFL++ Version	LLVM Version	Deterministic on by Default	Scheduler	CmpLog	mOpt	Dict2File
2.52c	8	✓	EXPLORE	✘	✘	✘
2.61c	11	✓	EXPLORE	✓	✓	✘
2.68c	11	✓	EXPLORE	✓	✓	✘
3.00c	11	✘	FAST	✓	✓	✓
3.14c	13	✘	FAST	✓	✓	✓
4.00c	13	✘	FAST	✓	✓	✓
4.04c	15	✘	FAST	✓	✓	✓
4.10c	17	✘ *new*	EXPLORE	✓	✘ *bug*	✓
4.21c	18	✓ *new*	EXPLORE	✓	✓	✓
4.32c	18	✓ *new*	EXPLORE	✓	✓	✓

new: new algorithm with same name; *bug*: AFL++ impl. bug, fixed in *4.20c*

Target Selection. To assess the evolution of AFL++ and its features, we selected a diverse set of open-source projects from the FUZZBENCH project [13]. These targets represent various domains and input structures, providing a comprehensive evaluation base. FUZZBENCH ensures consistent evaluations by running targets and fuzzers in Docker containers with a standardized build setup. We developed a custom testing environment to better control settings like Clang version and AFL++ configurations, which are difficult to adjust in FuzzBench. This environment allowed us to configure different AFL++ versions while still using FUZZBENCH and OSS-Fuzz's prebuilt Docker files for target-harness setup.

The selected targets cover diverse application domains, including binary analysis (BLOATY and LIBPCAP), image/media processing (LIBJPEG and OPENH264), network communication (CURL), and cryptographic operations (OPENSSL). We also chose targets that challenge AFL++'s grammar fuzzing, such as LIBXSLT and SQLITE3. Existing FUZZBENCH reports [32] helped prioritize targets with varied fuzzing results, such as HARFBUZZ, and recommendations from the AFL++ maintainer [33] led to the selection of targets like STB and FREETYPE2. Furthermore, BLOATY, HARFBUZZ, LIBAOM, LIBXML2, ASSIMP, and MRUBY are additionally used for our bug finding experiment. Overall, our main experiments are based on 10 of these 15 targets.

Fuzzing Feature Selection. AFL++ has significantly evolved over the years through community and academic contributions, adding features that improve fuzzing efficiency, instrumentation, and target compatibility. Notable additions include new mutators such as `mOpt`, advanced

instrumentation such as `CmpLog`, and extended binary support via `QEMU` and `Frida`. Testing all features would require immense computational resources.

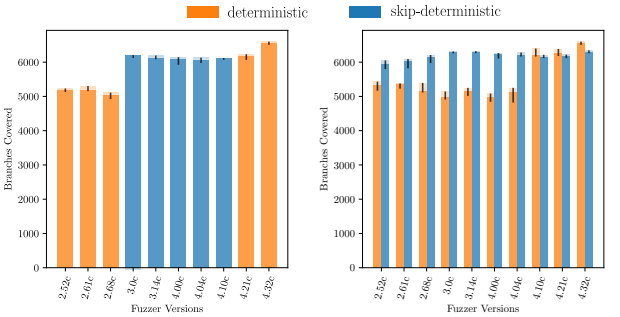
Our study analyzes the impact of selected features for general-purpose, source-based fuzzing, excluding those aimed at closed-source binaries (e.g., QEMU-mode or snapshot-based fuzzing with Nyx [28]). All tested features were evaluated in isolation to measure their individual contributions; combining multiple of them may yield better performance in practice. Furthermore, we used the default settings for each feature. Some can be fine-tuned via command-line options, but we did not explore parameter tuning as part of our evaluation.

To enable a focused yet meaningful analysis, we selected ten AFL++ features, grouped into two categories (fuzzing- and instrumentation-related) for a targeted evaluation. The selected fuzzing features include the default configuration, the deterministic fuzzing mode (enabled/disabled), and the default scheduler, which varies between `Explore` and `Fast` depending on the AFL++ version. We also evaluate the impact of the `mOpt` mutator, the usefulness of automatically extracted dictionaries (`dict2file`), the comparison logging mechanism (`CmpLog`), and the enhanced comparison handling feature (`CompCov`). These were selected to reflect key functional improvements and to capture how their implementation and effectiveness have evolved across AFL++ versions. Beyond these, AFL++ is continuously updated with potential enhancements, but such changes are not always explicitly configurable or directly user-controllable.

Instrumentation Mode Selection. While fuzzing features define how inputs are mutated, instrumentation features determine how coverage feedback is collected. Instrumentation is a critical component of feedback-driven fuzzing, enabling the detection of newly reached program states, guiding mutations, and ultimately maximizing code coverage. Over the years, AFL++ has integrated various instrumentation techniques with differing trade-offs in terms of precision, performance, and compatibility. For our analysis, we selected the following instrumentation modes: AFL++ `PCGuard`, `Classic`, `CTX`, and `llvm-native`, all described in detail in Section 2.

Target Instrumentation. To ensure a fair comparison across AFL++ versions, we used a hybrid instrumentation approach. Each target was instrumented in two ways: (1) using a recent AFL++ version and compiler, and (2) using each version's original instrumentation and compiler. This reduces external influences, such as compiler optimizations, isolating improvements in fuzzing strategies. Evaluating all versions with the modern setup highlights changes in fuzzing logic, while using the original configurations captures progress in instrumentation over time.

AFL++ *4.20c* introduced a new forkserver model that is incompatible with older versions. To address this, we instrumented versions older than *4.20c* (starting from *4.10c*) with AFL++ *4.10c*, while versions *4.20c* and newer were instrumented with AFL++ *4.21c* to ensure compatibility. Since version *4.30c* modifies the `CmpLog` map, requiring targets to be recompiled, we compiled the targets in the `CmpLog` experiment with the respective version under test (*4.32c*). The compiler version remained constant at LLVM *18.0*. This ensured all fuzzers used the same instrumented



(a) Default configuration (b) Effect of deterministic stage

Figure 2: Results for different AFL++ versions on BLOATY. The left plot shows the versions in their evolving *default* configurations, as the deterministic-mutation defaults changed over time. The right plot compares runs with the deterministic stage enabled versus skipped.

binary, eliminating discrepancies. For experiments comparing different instrumentation modes, we used both the latest and native (i.e., version-matched) setups to assess the impact of instrumentation and compiler optimizations.

Baseline Configuration. During initial testing, we noticed a significant improvement in most versions of AFL++ when skipping the deterministic stage (see Section 4.1), except those with the *new* deterministic stage (≥*4.10c*). Thus, to ensure an isolated evaluation of different features, each experiment—unless otherwise stated—was conducted with the deterministic stage being skipped.

AFL++ employs different execution strategies to optimize fuzzing speed and efficiency. The *forkserver* model reduces process startup overhead by keeping the process in memory and forking new instances when needed. The *persistent mode* further minimizes overhead by executing multiple fuzzing iterations within the same process, avoiding costly forks. Since the *persistent mode* offers faster execution and our target selection was based on FUZZBENCH targets (which already support it), all experiments used this strategy. Additionally, the *shared memory test case* feature, which passes test cases via shared memory to the target application, was enabled by default in the FUZZBENCH harnesses, providing further speedup.

4. Experimental Evaluation

We now discuss our four experiments and two case studies, which evaluate fuzzer improvement, instrumentation, long-term performance beyond 24 hours, and bug finding. Afterward, we discuss lessons learned.

4.1. Experiment 1: Fuzzer Improvement

We first evaluate the individual contributions of key features introduced in AFL++ over time. As described before, all targets are instrumented using the version (*4.10c* or *4.21c*) to minimize the influence of the compiler, as described in the previous section. We grouped the different features into two categories: instrumentation and feedback enhancements (`CmpLog`, `CompCov`, and `dict2file`), and mutation and scheduling strategies (`mOpt`, `Fast`, and `Explore`). As a baseline, we first evaluate the *default* configuration for each version to provide a reference point.

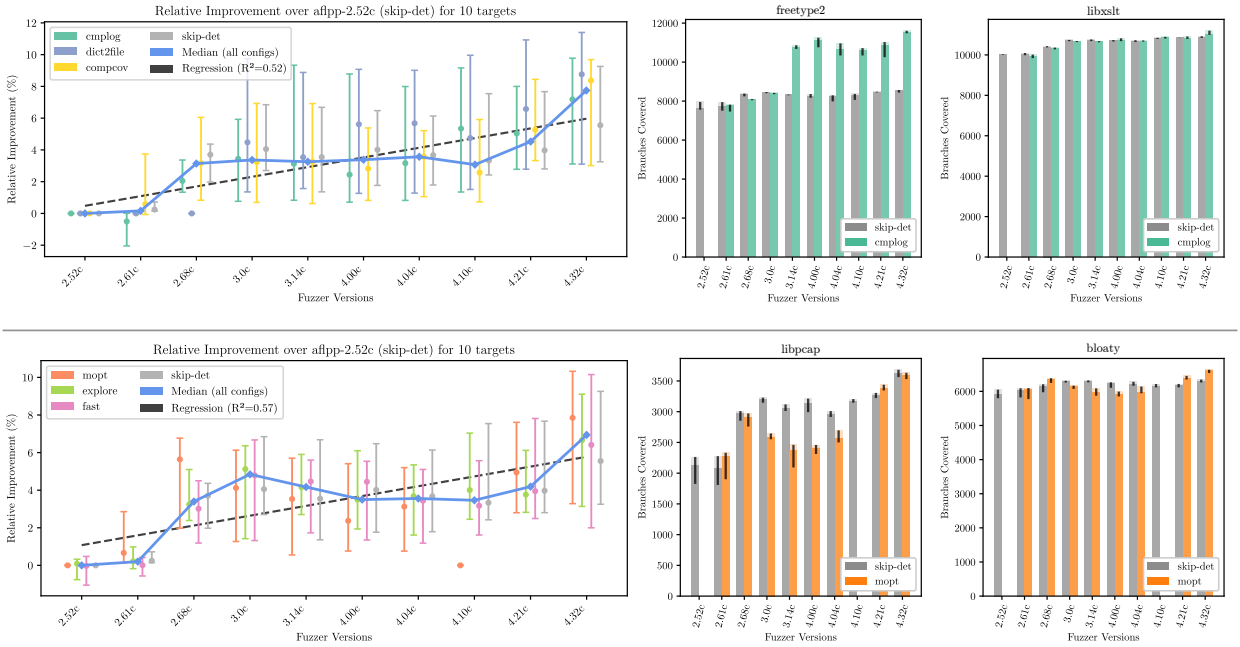


Figure 3: The plots on the left show the relative improvement with interquartile range (IQR) (Q1-Q3) as error bars of different versions and features (instrumentation/feedback enhancements on top, mutation/scheduling features bottom plot) over AFL++ *2.52c* (`skip-det`); results are aggregated over all ten targets. The median and linear regression, calculated over all targets and features, are shown as well. Bar charts on the right show how `CmpLog` (top) and `mOpt` (bottom) compare to the baseline `skip-det`, each for two representative targets.

Default Configuration. In our first experiment, we executed different AFL++ versions using their default configurations. This evaluation revealed a notable increase in code coverage for seven targets between versions *2.68c* and *3.0c*. Picking BLOATY as an example, the median coverage increased by 19.3% (see Figure 2a). This trend was also observed in our earlier exploratory experiment that included *all* AFL++ versions (see Figure 1). An analysis of the release notes highlights several changes introduced in version *3.0c*, including two major ones: (1) the default scheduler was changed from `Explore` to `Fast`, and (2) the deterministic stage was disabled.

To investigate whether these changes are causing the observed effects, we ran a second experiment testing each selected version with and without the deterministic stage. The results confirmed that disabling the deterministic stage was the primary contributor to the observed performance improvement. Figure 2b displays the differences between these two settings. The left plot shows the results for BLOATY using the default configuration, where the settings for the deterministic stage vary across versions. The right graph depicts the same target with fixed settings for the deterministic stage. Overall, the improvement from skipping this stage ranges from +24% for *3.0c* to -4.03% for *4.32c*. This suggests that the overhaul of the deterministic stage in version *4.10c* had a positive impact on this version and its successors. Since skipping the deterministic stage significantly improved performance across most versions, we disabled it for our experiments to isolate its impact.

Instrumentation & Feedback Enhancements. Due to the many combinations of features across ten targets and ten AFL++ versions, we present a consolidated summary in Figure 3, where each error bar and its corresponding markers represent all ten targets. The plot shows the *relative improvement* per version over AFL++ *2.52c* with the deterministic stage disabled (`skip-det`). Since aggregating across all targets introduces some variance, we computed a linear regression to capture the overall trend. The regression indicates a slight upward trend, with a slope of 0.61% per version and an R^2 of 0.52. This value reflects that much of the variance stems from target-specific differences rather than version progression itself, which is why the improvement pattern is not strongly linear. As the median line shows, the overall improvement stalls after version *2.68c*, remaining between 2% and 4%, and increases again after version *4.10c*.

Analyzing the features shows that most yield noticeable performance improvements up to version *3.0c*. While `dict2file` and `CmpLog` exhibit small performance gains after *3.0c*, `CompCov` experiences a decline after *3.14c*, followed by a recovery starting in version *4.10c*, ultimately reaching the best median performance (8%) for *4.32c*.

To further illustrate these trends, we selected two targets as case studies for `CmpLog`: one where it excels (FREETYPE2) and one with only average improvement (LIBXSLT), shown in Figure 3. When introduced in version *2.61c*, `CmpLog` even reduced coverage (-3.2% vs. `skip-det`), but from version *3.14c* onward it shows clear benefits, adding a median increase of 2,788 branches (+32.8%). Updates between versions *3.0c* and *3.14c*, such as handling floating points and transformations (e.g., *toupper*, *tolower*, *tohex*), account for this gain (see Appendix C). However, from this version onward, performance plateaus between 10,200 and 10,700 covered branches, showing no further substantial improvements until version *4.32c*. This version shows a median coverage of about 11,200 branches. The improvement stems from a fix of the `CmpLog` map in version *4.30c*, according to

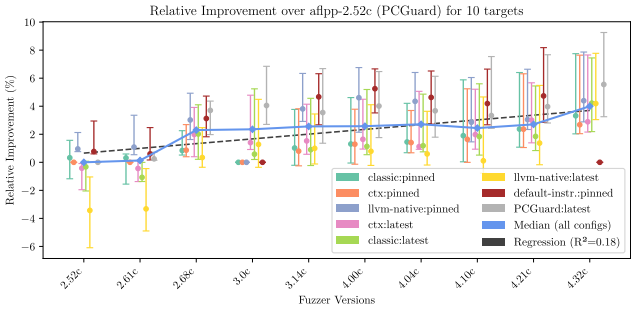


Figure 4: Relative improvement over time with AFL++ *2.52c* (`PCGuard`) as baseline over all targets and all instrumentation types with IQR as error bars. The median and linear regression are also shown.

the release notes. The overall significant improvement of `CmpLog` is specific for FREETYPE2, whereas for other targets, the effect is more marginal (0.1% to 3.04% over `skip-det`), as shown in Figure 3 for LIBXSLT.

Mutation & Scheduling Strategies. We also compared different scheduling and mutation strategies, as shown in Figure 3 (bottom plots). Unlike instrumentation and feedback enhancements, whose performance trends generally show steady improvement across versions, their performance trends are less consistent, as reflected by the median line. As the median indicates, after the same initial positive effect observed for the instrumentation and feedback features up to version *2.68c*, overall performance declines until version *4.10c* and then recovers. The linear regression shows only a minor upward trend, with a slope of 0.52%, which is even smaller than the trend observed for instrumentation and feedback enhancements.

The first tested version incorporating `mOpt` (*2.61c*) shows a slight positive effect compared to `Fast`, `Explore`, and `skip-det`. Starting from version *2.68*, a more pronounced improvement of 2% to 8% over the baseline *2.52c* (`skip-det`) can be observed. However, the median improvement of `mOpt` itself is only marginal (2%) compared to the `skip-det` configuration of the same version (*2.68c*). From that point onward, `mOpt`'s performance declines until version *4.00c*, after which it begins to recover. A similar trend is observed for the `Fast` power schedule, which peaked at *4.32c*, whereas the second tested power schedule, `Explore`, maintains more stable performance across all versions beyond *2.68c*. To further analyze these trends, we examined `mOpt` in detail using the two targets LIBPCAP and BLOATY. As Figure 3 illustrates, the decline and subsequent recovery of `mOpt` are evident on both targets, with the most recent version outperforming the baseline `skip-det` (grey bar) in both cases. Notably, a bug introduced in AFL++ *4.09c* temporarily prevented the use of `mOpt`, but according to the release notes of version *4.20c*, this issue was resolved.

Takeaway #1: Instrumentation and feedback enhancements yield small but steady gains across versions, with notable improvements up to AFL++ *3.0c*. In contrast, mutation and scheduling strategies exhibit more variable, fluctuating performance.

4.2. Experiment 2: Instrumentation

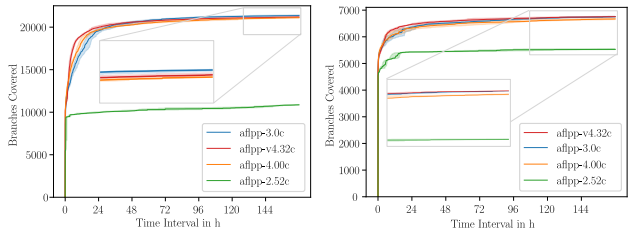
For this experiment, we analyze the impact of different instrumentation types in two scenarios: (1) targets instrumented using a recent version of the fuzzer and compiler (suffix *latest*), as in the previous experiment (cf. Section 4.1), and (2) targets instrumented using the version of the fuzzer/compiler combination corresponding to each release (suffix *pinned*). As described in Section 3, the instrumentation types under evaluation include `Classic`, `llvm-native`, `CTX`, and the default instrumentation of AFL++ version *4.21c*, `PCGuard`. The results presented in Figure 4 show a slight visible improvement in terms of coverage beyond the selected baseline (AFL++ *2.52c* with `PCGuard`) that cannot be attributed to general fuzzer improvements observed in the previous experiment. In nearly all versions, the instrumentation with the version-specific compiler (*pinned*) yields slightly better performance than using a recent compiler version. To illustrate this trend, we fit a linear regression and plot the combined median across all targets and instrumentation types. The regression yields a slope of 0.34% and an R^2 of 0.18, indicating only a marginal upward trend across versions, with the low R^2 reflecting that most variance originates from target-specific performance differences rather than version progression. This experiment also exhibits considerable variance, likely due to the different types of target programs. A plot including values outside the interquartile range is provided in Appendix C.

Takeaway #2: Different instrumentation types have little to no impact on coverage when compared to the default `PCGuard` instrumentation used in the previous experiments (represented by the grey bar).

4.3. Experiment 3: Long-Term Experiment

Beyond our standard 24-hour experiments, evaluating longer runs is valuable, particularly for complex targets that do not reach coverage saturation within a day. Therefore, we selected a subset of AFL++ versions (*2.52c*, *3.0c*, *4.0c*, and *4.32c*) and two targets (BLOATY and SQLITE3) for a 7-day campaign. These targets were chosen to represent different input characteristics: SQLITE3 requires structured input, while BLOATY processes binary input. For this experiment, we selected the unaltered default configuration for each AFL++ version, resulting in some fuzzers (e.g., *2.52c* and *4.32c*) using the deterministic stage, while others (e.g., *3.0c* and *4.0c*) skip it.

The results are shown in Figure 5. On both targets, version *4.32c* initially explores the program more efficiently, achieving greater coverage within the first 48 hours compared to *3.0c* and *4.0c*, while *2.52c* significantly lags behind, reaching only about 50% of the coverage achieved by the other versions. For SQLITE3, shortly after the 24-hour mark, *3.0c* catches up. After 48 hours, version *3.0c* surpasses the others, maintaining a slight advantage over the whole seven-day period, with a median coverage of 21,359.5 branches compared to 21,186.5 for *4.32c*. A similar trend is observed for BLOATY. Here, version *3.0c* overtakes *4.0c* shortly before the 24-hour mark and nearly matches *4.32c* by the end of the seven days, ending with a negligible advantage for *4.32c* (6,756.5 vs. 6,752.5



(a) SQLITE3 (b) BLOATY

Figure 5: Median coverage of 10 trials over seven days in the default configuration for SQLITE3 and BLOATY. On SQLITE3, AFL++ *3.0c* (*21,359*) shows a slight advantage over the latest version, *4.32c* (*21,186*), and it performs similarly on BLOATY (*6,752* vs. *6,756*).

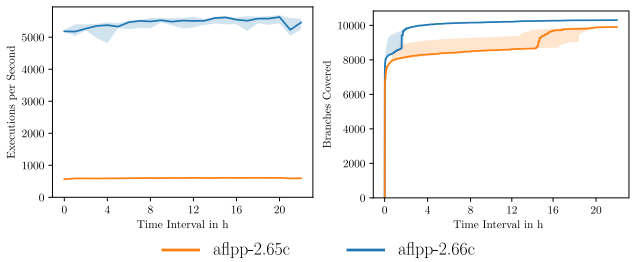


Figure 6: Case study of AFL++ *2.65c* and *2.66c* on LIBXSLT, highlighting the speedup of the *shared memory test case* feature. Left side shows the median executions per second, right plot shows coverage (along with the 25th and 75th percentiles over 10 repetitions).

covered branches). An interesting observation is that both versions *3.0c* and *4.0c* use the same default seed schedule configuration (`Fast`) and both skip the deterministic stage; yet, *3.0c* consistently outperforms *4.0c*. This aligns with our previous experiments, where we also observed that *3.0c* can surpass more recent AFL++ versions. However, as explored in a case study in Appendix C, we could not precisely identify the specific code changes in subsequent AFL++ versions that caused this regression.

One possible explanation for *why* this regression, which ultimately gives version *3.0c* a long-term advantage, went unnoticed is that new AFL++ features are typically evaluated using FUZZBENCH in the Google Cloud. Because of technical constraints, these campaigns are limited to 23 hours of execution, which prevents regressions that appear only in longer runs from being detected.

Takeaway #3: While the latest tested AFL++ version is more effective in the first 24 hours, *3.0c* surpasses it on one target and nearly matches its performance on the other over the seven-day run.

4.4. Experiment 4: AFL++ Bug-finding

The primary metric for measuring fuzzer performance in our paper is branch coverage, following the insights of Böhme et al. [11]. However, coverage is ultimately only a proxy for the true objective of fuzzing: finding bugs. To evaluate this aspect directly, we conducted a bug-finding experiment on six FUZZBENCH targets. These targets were selected from the default FUZZBENCH set

TABLE 2: Unique bugs found by different versions across six FUZZBENCH targets (24h; 10 runs each; `skip-det`).

Fuzzer	HARFBUZZ (17863bd)				BLOATY (52948c1)				LIBXML2 (e85f9b9)			
	Min.	Med.	Max.	Avg.	Min.	Med.	Max.	Avg.	Min.	Med.	Max.	Avg.
2.52c	2	3.5	5	3.3	1	1.0	1	1.0	0	0.0	1	0.2
3.0c	2	4.0	8	3.8	1	1.0	1	1.0	2	2.0	2	2.0
4.00c	2	4.0	7	3.8	1	1.0	1	1.0	2	2.0	2	2.0
v4.10c	2	4.0	6	3.8	1	1.0	1	1.0	0	0.0	1	0.1
v4.21c	2	4.0	4	3.3	1	1.0	1	1.0	0	0.0	2	0.4
v4.32c	2	4.0	8	4.1	1	1.0	1	1.0	2	2.0	2	2.0
v4.32c (SAND)	4	6.0	7	5.7	1	1.0	1	1.0	2	2.0	2	2.0
	ASSIMP (4d451fe)				MRUBY (8c8bbd9)				LIBAOM (6e18489)			
	Min.	Med.	Max.	Avg.	Min.	Med.	Max.	Avg.	Min.	Med.	Max.	Avg.
2.52c	0	0.0	2	0.2	0	0.0	0	0.0	0	0.0	2	0.4
3.0c	0	0.0	4	0.4	0	0.0	0	0.0	0	3.0	3	2.8
4.00c	0	0.0	0	0.0	0	0.0	1	0.1	3	3.0	3	3.0
v4.10c	0	0.0	0	0.0	0	0.0	0	0.0	3	3.0	3	3.0
v4.21c	0	0.0	0	0.0	0	0.0	0	0.0	0	3.0	3	2.6
v4.32c	0	0.0	6	1.5	0	0.0	1	0.3	3	3.0	3	3.0
v4.32c (SAND)	0	0.0	0	0.0	0	0.0	0	0.0	3	3.0	3	3.0

(HARFBUZZ, BLOATY, LIBXML2, and MRUBY) as well as from public FUZZBENCH bug experiments (LIBAOM and ASSIMP). The targets are compiled and fuzzed with the same flags and sanitizers as done by FUZZBENCH. We deduplicate crashes by parsing each sanitizer report, extracting the bug type and the originating source code location within the project's source tree. Each unique pair of bug type and source location represents a unique bug.

The results of this experiment, summarized in Table 2, highlight several interesting patterns: For most targets (BLOATY, LIBXML2, and MRUBY), the number of unique bugs remains very low and is often constant across all AFL++ versions. In contrast, the target HARFBUZZ shows greater variation. While recent versions such as `v4.32c` and its `SAND` variant achieve the highest averages and maxima, the older `3.0c` release still exposes certain bugs that later versions miss. Interestingly, version *4.32c* with SAND does not find bugs in ASSIMP and MRUBY, but *4.32c* without SAND and older versions, such as *3.0c* (ASSIMP) and *4.0c* (MRUBY), can trigger bugs. On LIBAOM, nearly all versions cover the same three bugs.

Takeaways: The low number of bugs, paired with often negligible differences between fuzzer versions, makes bug experiments unsuitable for fine-grained measurement of progress across fuzzer versions.

4.5. Case Study: Improvement 2.61c – 2.68c

Starting with AFL++ versions *2.61c* and continuing through version *2.68c*, we observed a notable improvement in coverage across most targets and configurations (cf. Figure 3). A closer examination revealed a significant difference in the total number of executions between these two versions. For instance, for FREETYPE2, version *2.68c* achieved approximately 560% more executions than *2.61c*, a trend we also observed on other targets. To identify the specific improvement responsible for this increase in executions, we first examined the intermediate versions and conducted test runs to narrow down the relevant changes. We determined that the improvement occurred between versions *2.65c* and *2.66c* (see Figure 6). According to the release notes, several changes could have contributed to this improvement. However, the most likely candidate was the introduction of the *shared memory test case* feature, which enables passing of the test case

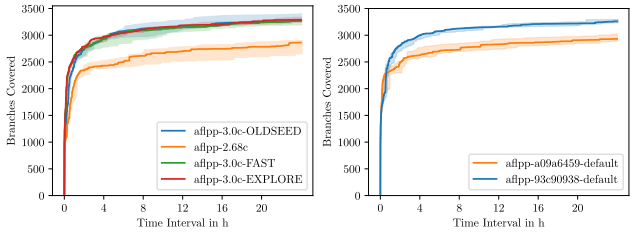


Figure 7: Case study of the improvement from *2.68c* to *3.0c* on LIBPCAP. The left shows changes to default settings had little effect, the right shows the commits before (orange) and after (blue) the havoc mode expansion.

via shared memory rather than using standard input or a file. This hypothesis was confirmed by an experiment comparing the commit immediately before the feature was introduced with the commit that added it.

> **Takeaway #4:** The *shared memory test case* feature significantly accelerates fuzzing, increasing coverage across all targets.

4.6. Case Study: Improvement 2.68c – 3.0c

With the move from *2.68c* to the next release, *3.0c*, we observed an improvement in branch coverage. While the difference is minor for most targets, it is more pronounced for LIBPCAP, BLOATY, and LIBXSLT. According to the release notes, several default settings changed in *3.0c*, including switching the default seed schedule from `Explore` to `Fast`, and introducing a new seed selection mechanism based on random weights rather than a sequential approach. To evaluate the impact of these changes, we conducted an experiment on the LIBPCAP target, comparing different schedules and the new seed selection mechanism. The results, shown in Figure 7, indicate that these modifications did not significantly affect performance for this target. This experiment suggests the performance gains were not solely due to noticeable configuration changes. To investigate, we bisected the commits between versions *2.68c* and *3.0c* and reran the campaign. We identified a minor yet significant change to six lines of source code in `afl-fuzz-one.c`: if no new findings happen within five seconds, the *havoc* mode performs more expensive mutations. The right plot in Figure 7 illustrates the effect by comparing the commits before and after this change.

> **Takeaway #5:** Minor changes can have a significant impact. Expanding the *havoc* stage in version *3.0c* positively affected program exploration. For LIBPCAP, this results in a median improvement of 11.2%.

4.7. Lessons Learned

Based on all the experiments conducted in this study, we summarize our observations into four key insights:

L1 – Progress largely stalled. Coverage improved until *3.0c*, then largely plateaued (cf. Figure 3), with a slight improvement due to *4.10c*'s new deterministic stage.

L2 – Few features yield generic improvements. Only a few features, most reliably `dict2file`, `CmpLog`, `CompCov`, and the *new* deterministic stage, lead to repeatable gains across diverse targets. Most other features provide benefits only in isolated cases.

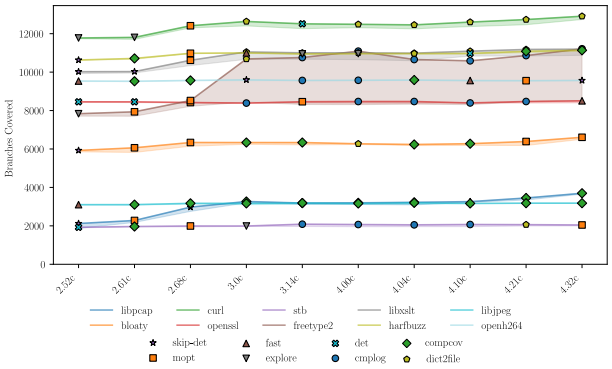


Figure 8: Each symbol represents the *best* of all configurations for this AFL++ version and target. The interval highlights the coverage difference to the median configuration (mostly relevant for FREETYPE2, where the best configuration is significantly better than the median one).

L3 – The largest gains stem from simple features. We observed that relatively small changes, such as the extension of the havoc mode, using shared memory for test case transfer, and disabling the deterministic stage (*2.68c*→*3.0c*), delivered some of the most significant performance improvements.

L4 – Improvements rarely generalize across targets. What works well on targets such as FREETYPE2 or CURL does not translate to similar improvements for other targets such as OPENSSL and STB. Figure 8 (and Table 5 in the Appendix) track the best-performing configuration per target, showing a large variety across targets and versions.

With the empirical history of AFL++ mapped out, a new research question surfaces: *How readily do new ideas from the academic fuzzing community make their way into practice?* After all, while these works often propose sophisticated improvements, our empirical observations of low coverage progress stand in contrast to the large number of research papers published in this domain. We will study this question in Section 5.

4.8. Generalizability of Observations

One key question is whether our observations, particularly *L1*, are specific to AFL++ or indicative of a broader trend. To test this hypothesis, we include the *only* other general-purpose fuzzer with active development and research integration, LIBAFL [34]. Second, we examine if our observations hold for industry-backed fuzzers. Unfortunately, such fuzzers are not generally open source, with one notable exception: FUZZILLI [35]. It is considered state of the art in JavaScript fuzzing and is actively maintained by Google engineers [36].

4.8.1. Experiment 5: LIBAFL. We study whether our observations hold for LIBAFL, a recent alternative to AFL++ designed as a successor of the deprecated LIBFUZZER. Because LIBAFL is intended as a fuzzing library rather than a full fuzzer, it does not provide a complete fuzzer implementation or default configurations.

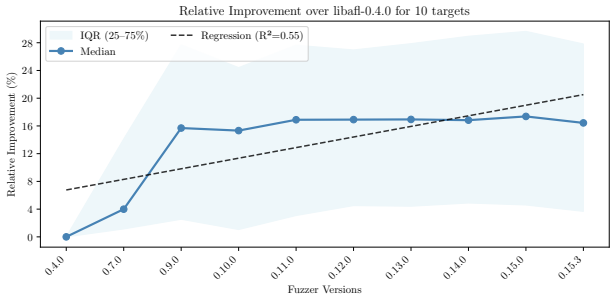


Figure 9: Median relative improvement and linear regression per released LIBAFL version over 10 targets and 10 repetitions. The plot connects medians to show trends, not continuous values.

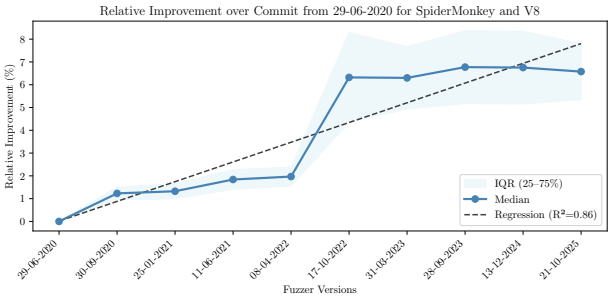


Figure 10: Median relative improvement and linear regression per FUZZILLI commit over the targets SPIDERMONKEY and V8 and 10 repetitions. The plot connects medians to show trends, not continuous values.

Experiment Setup. LIBAFL has, at the time of writing, 27 release versions, starting in April 2021. We selected 10 versions across the development cycle of the fuzzer, beginning with version *0.4.0* from 2021, which is the first version that provides an in-process harness for FUZZBENCH targets. This harness allows us to use the same target set as in our AFL++ experiments, facilitating comparability. According to the LIBAFL Git repository, this harness uses the best possible settings for the fuzzer.

Results. Figure 9 shows the improvement of the LIBAFL harness across the target set. We plot the median improvement together with the 25th–75th percentile range relative to the first selected version (*0.4.0*) across all ten targets. As the median improvement indicates, the major gain occurs between versions *0.4.0* and *0.9.0*, with a median increase of 16% and the 75th percentile reaching 25%. Thereafter, the median stabilized at approximately 16%. A version of the plot including values outside the IQR is provided in Appendix 15.

The steep early improvement can be attributed to several factors, including the integration of well-established techniques from AFL++ (such as `CmpLog` (*0.4.0-0.9.0+*), `mOpt` (*0.5.0-0.8.0*), and improved power schedules (*0.6.0*)) as well as various bug fixes in the fuzzer. To indicate the overall trend, a linear regression was fitted to the median relative improvements across the versions. The resulting model exhibits a moderate positive trend (an increase of 1.53% in relative improvement per version), with an R^2 of 0.55. This value reflects that a substantial portion of the variance arises from systematic differences between targets, not solely from version progression, meaning that while newer versions generally improve coverage, the gains are not strictly linear and vary across releases.

4.8.2. Experiment 6: FUZZILLI. Potentially, our observations may be influenced by the fact that these fuzzers are maintained by an open-source community. To address this concern, we additionally include an industry-maintained fuzzer: FUZZILLI. We opt for this JavaScript fuzzer, as the industry does not actively maintain an open-source, general-purpose fuzzer. However, FUZZILLI is actively maintained and used to continuously fuzz the JavaScript Engines of Mozilla's SPIDERMONKEY and Google's V8.

Experiment Setup. We selected 13 FUZZILLI commits spanning the fuzzer's development timeline. For our evaluation, we chose the latest versions of the JavaScript engines used in Chromium (V8) and Mozilla (SPIDERMONKEY). Over time, various features have been integrated into the FUZZILLI codebase, including new mutators and reducers, extended JavaScript language support, and several minor optimizations to engine-specific profiles. Unlike AFL++, most of these features are enabled by default, eliminating the need for manual pre-selection.

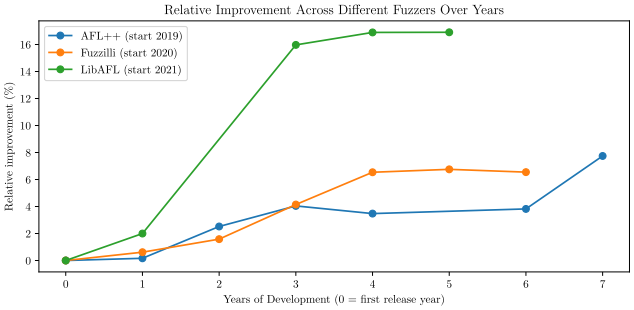


Figure 11: Median relative improvements of AFL++, LIBAFL, and FUZZILLI. The x-axis represents the relative age of the tested versions in years, with 0 indicating the earliest version of each fuzzer included in our study rather than the first version ever released.

Results. Figure 10 shows the results of our FUZZILLI experiments. As the plot indicates, the achieved coverage increases across the selected commits. As in the previous experiments, we plot the median improvement together with the 25th–75th percentile range over both targets. In the first two years of development, the median shows a small but steady improvement to around 2%, followed by a substantial jump to approximately 6.5% in 2022, after which the trend stagnates. The regression has a slope of 0.87% per version and an R^2 of 0.86, indicating a consistent upward trend and that the model explains most of the variance across versions.

Takeaways: Our observations, in particular L1 and L4, also apply more generally to LIBAFL and FUZZILLI. Figure 11 shows all improvements over time: LIBAFL's initial strong growth benefits from the initial adoption of proven techniques.

5. Meta-Study of Scientific Literature

Our evaluation of AFL++'s chronological development reveals stagnation over the past four years, with a slight improvement in the most recent versions. During the same period, many fuzzing papers on general-purpose fuzzing have been published. This discrepancy raises a natural research question: *How readily do new ideas from the academic fuzzing community make their way into practice?* To answer this question, we conducted a literature survey of fuzzing papers published over the past years. Our goal was to determine whether these works have been integrated into AFL++ and, if not, to understand *why*. We first outline the methodology and then discuss the results.

5.1. Methodology

We examined all fuzzing papers published in top security (USENIX Security, S&P, CCS, and NDSS) and software engineering venues (ICSE, FSE, and ASE) from 2018–2024. We *included* papers whose primary contribution is a fuzzing technique (e.g., a new scheduler, mutation strategy, or coverage metric). We *excluded* papers that merely *use* fuzzing to support another line of work (e.g., generating diverse inputs). Our search produced 405 candidate papers. To gauge practical integrability, we further filtered for works that already build on AFL++ or its predecessor AFL, since basing on a shared code base lowers the integration barrier. However, we note that outliers like REDQUEEN, which was integrated despite its implementation being based on kAFL [37], exist.

Under these assumptions, we identified 44 AFL-based and 16 AFL++-based papers (cf. Table 3). For practical reasons, we consider only works that provide publicly available source code, as this is crucial for the further adoption of the technique. For the remaining 44 publications, we used the metrics described below to determine their degree of integrability into upstream AFL/AFL++.

Integration Metrics. Integration feasibility depends on:

- **Setup complexity**: Quality of documentation and dependence on external tools.
- **Code-base compatibility**: Fits into AFL++ mode/pass/mutator with minimal code changes.
- **Benefit vs. complexity**: Are there statistically significant gains that justify the added complexity?
- **Generality**: Does the approach's domain match the use case of AFL++, i.e., does it follow the principles of general-purpose fuzzing?

To apply these metrics, we examined both the paper and any publicly available artifacts (e.g., GitHub repositories). For instance, to estimate code complexity, we checked whether the authors reported the amount of code added on top of AFL or AFL++ and inspected the artifact in more detail. If the paper did not specify the extent of code changes, we cloned the repository and generated a diff against the baseline version. However, this does not apply to every paper: some only provide partial code, others omit the exact version or commit of the fuzzer, or the code is released solely as Docker images or FUZZBENCH integrations, further complicating the process.

TABLE 3: Papers based on AFL or AFL++. Greyed-out references indicate papers without public source code. The last row summarizes *#papers with source / #total papers*.

Year	Based on **AFL**	Based on **AFL++**
2018	[38], [39], [40]	–
2019	[41], [23], [42], [43], [44], [45], [46], [47]	–
2020	[48], [49], [50], [51], [52], [53], [54], [55], [56], [57]	–
2021	[58], [59], [60], [16], [61], [62], [63]	[15], [64]
2022	[65], [66], [67], [68], [69], [70], [71]	[17], [72]
2023	[73], [74], [75], [76], [77], [78]	[20], [79], [19], [80]
2024	[81], [82], [83]	[84], [85], [21], [86], [87], [88]
2025	–	[89], [90]
AFL src available: 31/44	**AFL++ src available:** 13/16	**Total:** 44/60

Yellow background: integrated in 2024 but published after our literature analysis

Based on this manual review, we assigned a score estimating its likelihood of integration into AFL++. While the checklist provided a structured way to assess each work, the final scores relied on our judgment, informed by our experience, the paper, and its artifact. We acknowledge that other researchers may arrive at a different score.

5.2. Results

Our literature review identified 44 papers, and we applied our integration scoring scheme to each of them. As shown in Table 3, AFL reached its peak in 2020, but since 2021, AFL++ has been used more frequently. Interestingly, some work published in 2024 is still based on the now-deprecated AFL. Our assessment results are summarized in Table 4. Using our scoring, we identified 17 papers as integrable, while the remaining 27 may be integrable, with varying degrees of feasibility. For the purpose of this analysis, we consider them as impractical for integration. We then contrasted this with AFL++'s integration efforts, observing four distinct outcomes:

Integration Impractical (27 Papers). These papers have goals or underlying assumptions that (significantly) deviate from AFL++'s general-purpose philosophy. Note that we considered Untracer [42] as impractical to integrate, but AFL++ has partially integrated it by providing a skeleton file for compatibility; thus, we do count it as *integrated*. We stress that this verdict does *not* imply the work is without merit; rather, its focus is orthogonal to AFL++'s core design. We observe four reasons for publications that are deemed too impractical for integration:

(a) **Domain-specific scope**, i.e., a narrow input domain or an "attack" class incompatible to AFL++'s "one-size-fits-all" [16], [17], [45], [52], [53], [60], [67], [77], [79], [86].

(b) **Heavy external requirements**, including cost-intensive analyses such as symbolic execution or full taint tracking that contradict AFL++'s lightweight forkserver model [39], [41], [49], [51], [69], [75].

(c) **Already subsumed by AFL++**, i.e., solved independently or via similar techniques (often for binary-only fuzzing) [50], [58], [59].

(d) **Directed or invariant-driven fuzzing** that steer testing towards specified locations or enforce invariants and, thus, pursue a goal too different from AFL++'s breadth-first coverage strategy [15], [43], [65], [76], [81], [82], [85].

TABLE 4: Papers based on AFL/AFL++, if we consider them to be integrable into AFL++, and the outcome thereof.

Paper	Year	Venue	Basis	Score	Integrated	Summary	Integration Notes
[39]	2018	USENIX	AFL	◕	✘	concolic execution	–
[38]	2018	ASE	AFL	●	(✓)	focus on rare branches	too complex, but inspired "rare scheduler" [91]
[41]	2019	CCS	AFL	◕	✘	hybrid fuzzing with taint-tracking	–
[42]	2019	S&P	AFL	◑	(✓)	binary-only fuzzing, coverage tracing	example skeleton provided
[23]	2019	USENIX	AFL	●	✓	swarm optimization mutations	integrated
[45]	2019	S&P	AFL	○	✘	file system fuzzing	–
[44]	2019	ICSE	AFL	●	(✓)	grammar fuzzing	named as third-party mutator in docs
[43]	2019	ICSE	AFL	○	✘	diff. fuzzing with Java decompilation	–
[52]	2020	NDSS	AFL	◑	✘	memory corruption focused	–
[49]	2020	S&P	AFL	◕	✘	hybrid fuzzing with concolic execution	–
[54]	2020	USENIX	AFL	●	✘	adaptive seed scheduling	results not reproducible [92]
[50]	2020	S&P	AFL	◑	✘	binary rewriting	–
[53]	2020	ICSE	AFL	◕	✘	memory consumption feedback	–
[51]	2020	ICSE	AFL	◕	✘	hybrid fuzzing, differential testing	–
[48]	2020	S&P	AFL	●	✘	annotation guided fuzzing	limited practical usefulness [93]
[16]	2021	CCS	AFL	◑	✘	regression testing	–
[58]	2021	USENIX	AFL	◑	✘	binary only fuzzing	–
[59]	2021	CCS	AFL	◕	✘	binary only fuzzing	–
[15]	2021	USENIX	AFL++	◑	✘	likely invariant fuzzing	–
[60]	2021	CCS	AFL	○	✘	hypervisor fuzzing and virtualization	–
[64]	2021	NDSS	AFL++	●	✘	hierarchical seed scheduling	–
[17]	2022	CCS	AFL++	◑	✘	bug triaging	–
[65]	2022	ICSE	AFL	◑	✘	directed fuzzer	–
[68]	2022	ASE	AFL	●	✘	heap temporal bugs	–
[67]	2022	ICSE	AFL	◑	✘	LTL fuzzer	–
[69]	2022	FSE	AFL	◕	✘	symbolic execution	–
[66]	2022	NDSS	AFL	●	✘	history driven mutations	–
[74]	2023	NDSS	AFL	●	✘	evolutionary based mutations	no statistical significant improvements*
[20]	2023	USENIX	AFL++	●	✘	input prioritization / scheduling	limited improvement*
[73]	2023	CCS	AFL	●	✘	modeling of input processing logic	–
[79]	2023	USENIX	AFL++	◕	✘	multi language fuzzing	–
[75]	2023	ICSE	AFL	◕	✘	hybrid fuzzing	–
[19]	2023	ICSE	AFL++	●	✘	seed mutation strategy	–
[77]	2023	USENIX	AFL	◕	✘	program option fuzzing	–
[76]	2023	USENIX	AFL	◑	✘	directed fuzzing	–
[21]	2024	CCS	AFL++	●	✘	online stochastic control formulation	limited improvement* unless edge pruning disabled [94]
[84]	2024	NDSS	AFL++	●	✘	LLVM passes for context sensitivity	–
[87]	2024	NDSS	AFL++	●	✘	byte position / scheduling mutations	–
[86]	2024	USENIX	AFL++	○	✘	DBMS fuzzer	–
[85]	2024	USENIX	AFL++	◑	✘	directed multi-target fuzzing	–
[82]	2024	S&P	AFL	◑	✘	directed fuzzer	–
[81]	2024	S&P	AFL	◑	✘	directed fuzzer	–
[89]	2025	FSE	AFL++	●	✓	fast deterministic stage	integrated
[90]	2025	ICSE	AFL++	●	✓	reduce sanitizer overhead	integrated

Our assessment if techniques could be integrated: ● = integrable, ◑ = potentially integrable, ◕ = likely not integrable, and ○ = not integrable;
Gray background: integration unsuccessfully attempted by AFL++; *Yellow background:* integrated but published after our literature analysis;
*: assessment is based on our discussion with maintainers.

Integration Efforts Failed (6 Papers). Papers in this category [20], [21], [38], [48], [54], [74], highlighted with a gray background in Table 4, were considered by AFL++ maintainers as candidates for integration but were ultimately not included. Again, this does not imply a lack of merit. Our classification is based on public sources, such as GitHub issues or pull requests [91]–[93], and our direct communication with the AFL++ maintainers.

For instance, EcoFuzz [54] was initially promising, but its results could not be reproduced reliably [92]. FOX [21] was another integration candidate, but modifications such as disabling edge pruning [94] led to inflated coverage, thereby preventing it from achieving the claimed improvements. Furthermore, FairFuzz [38] achieved noticeable gains over the baseline, but its codebase changes were deemed too complex for full integration. Nevertheless, it inspired the development of a new rare-branch-based scheduling strategy AFL++ [91].

Potentially Integrable (7 Papers). We considered these papers to be integrable but they have not been picked up by AFL++ (yet). These works align closely with the design philosophy of AFL++, often by introducing new mutation strategies [19], [66], [73], [87] or feedback mechanisms [64], [68], [84]. While these works may appear promising on a technical level, they may not align with the practical priorities of the AFL++ developers. However, we cannot determine with certainty whether these papers were considered for integration or lacked the practical impact necessary to justify their adoption upstream.

Integrated into AFL++ (6 Papers). This category covers works that were fully [23], [89], [90] or partially merged [38], [42], [44] into AFL++. While the list includes only three fully integrated and three partially integrated papers, AFL++ incorporates a much broader range of features. Many of them originate from research not based on AFL or AFL++, or were not published in top-tier security or software engineering venues, thus being out of scope of our literature analysis. For a comprehensive overview, we refer the reader to Appendix 6, which tracks the origin of integrated features and techniques.

5.3. Lessons Learned

As discussed in Section 4.7, AFL++ versions released after version *3.0c* produced only modest, mostly target-specific improvements, with a more pronounced gain between *4.21c* and *4.32c*. Our meta-study illuminates *why* this plateau persisted for so long and continues to pose a

barrier: Although the fuzzing literature offers many new ideas, only a handful satisfy the practical constraints that lead to upstream adoption. Still, if adopted, fuzzing research can indeed improve the state of the art, as seen with the introduction of the new deterministic stage derived from MendelFuzz [89]. Finally, Figure 12 and Table 6 in the appendix show that the majority of features that are integrated and have proven successful do not necessarily originate from top-tier security and software engineering conferences, but also from workshops [95], journals [96], or individual and industry contributions [29], [97]–[100]. Furthermore, both the Table and Figure indicate that the focus of integrated works has shifted from the categories *features*, *improvements*, and *special use cases* to the latter over the past years. Overall, we draw three lessons:

L5 – Novelty and prestige are not enough. A technique's scientific merit or publication at a top-tier venue has little impact on adoption if it breaks AFL++'s forkserver model, abstraction layer, or demands heavyweight side engines. Conversely, many successful features originate from workshops, journals, industry, or individual contributions, demonstrating that practical integrability—not novelty or venue—is the primary determinant for adoption in practice.

L6 – Integrability over marginal gains. Papers that deliver modest but repeatable improvements and require small code changes (e.g., new mutators or lightweight schedulers) are far more likely to be merged than complex rewrites that promise double-digit speed-ups on a niche benchmark. Counterintuitively, it may be precisely the works that claim less "novelty" (and thus appear less appealing during peer review) that deliver improvements that move the needle in practice.

L7 – Reproducibility as a gatekeeper. Over one-third of the surveyed works provide no artifact, making integration and reproduction nearly impossible. For four of the six integration candidates, the hoped-for improvements outlined in the paper did not manifest when maintainers attempted to integrate them in AFL++. While the security community's adoption of artifact evaluation may improve artifact availability, current fuzzing evaluations often fail to capture how well their methods scale to new targets or different environments.

Our results indicate that reproducibility is key to bridging the gap between research and practice, enabling rigorous evaluation and cumulative scientific progress.

5.4. Practical implications

Over the course of our research, the following principles emerged as key factors toward practical impact:

- Favor low-complexity integrations over large architectural changes.
- Ensure reproducibility and artifact availability as a prerequisite for adoption.
- Demonstrate consistent gains across multiple targets, not only isolated improvements.
- Align with existing abstractions and execution models (e.g., forkserver-based fuzzing).

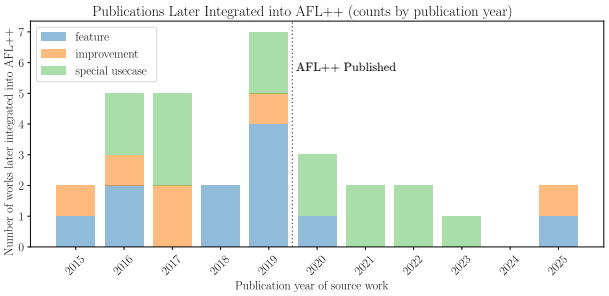


Figure 12: All techniques integrated into AFL++ in addition to the techniques analyzed in the meta-study. Bars indicate the year of paper publication (not integration).

We invite researchers to carefully reflect and consider them for future work. We intentionally provide no checklist that can be blindly applied by or even held against research, as these principles merely indicate a general tendency on how to achieve impact in practice. Their application requires careful consideration on a case-by-case basis.

6. Discussion and Limitations

We now reflect on the broader implications of our findings and limitations of our methodology and scope.

AFL++ as a proxy for research progress. We use AFL++ as a proxy for general-purpose fuzzing progress because it (i) serves as the de facto evaluation baseline in a large fraction of published work, (ii) continuously integrates research-driven techniques, and (iii) is widely used in both academia and industry. However, AFL++ does not capture all directions of fuzzing research. In particular, domain-specific and hybrid approaches may achieve significant improvements outside its design space. Our results should therefore be interpreted as reflecting progress within general-purpose coverage-guided fuzzing, rather than the entire fuzzing landscape.

Comparability of AFL++ and LibAFL While it is tempting to directly compare fuzzer results, our methodology of measuring improvement relative to the base configuration of each fuzzer does not allow such a comparison, as assumptions and usage scenarios differ between fuzzers. For example, AFL++ runs in a basic configuration on FuzzBench, while LibAFL maintainers opted to maintain an optimal harness. Thus, we cannot conclude if LibAFL is necessarily better than AFL++ beyond the specific, hidden set of assumptions underlying the respective configurations. LibAFL's initially steep improvement may also stem from an initial "ramp-up" phase, where they add proven features from other fuzzers and generally improve maturity. Branching from the already mature AFL, AFL++ had no such initial ramp-up phase.

Literature Study Completeness and Paper Filtering. We focused on A* software engineering and security venues [101], because they set our community's methodological bar (e.g., they typically enforce strict evaluation and reproducibility standards), but this excludes valuable ideas from smaller conferences, journals, and

workshops. We selected papers via keyword filtering, deduplication, and manual review, which streamlined the process but risks missing less-visible work. We restricted our analysis to AFL/AFL++-based tools to enable a consistent integration scoring, which may omit works such as REDQUEEN [24]. Future studies could broaden the scope to include other baselines and publication venues.

Integrability Scoring. We assessed whether fuzzing papers could be integrated into AFL++ by considering factors such as artifact availability, code complexity, build and dependency management, documentation quality, update history, and reported performance data. We have not compiled or run fuzzers, such that the scores represent high-level judgments that should be interpreted as indicative but not definitive verdicts.

7. Conclusion

In this paper, we presented a comprehensive evaluation of the evolution of AFL++ and examined the integrability of recent fuzzing research into this state-of-the-art fuzzer. Spanning seven years, our study shows that overall fuzzing effectiveness in terms of code coverage has largely plateaued. While individual changes can yield measurable gains on specific targets, their combined effect does not consistently scale across a diverse set of programs. Studying LIBAFL and FUZZILLI, we observe that this phenomenon is not limited to AFL++.

In the past, many general-purpose fuzzing papers have reported substantial performance improvements over their baselines, but double-digit coverage gains have become rare. When recent techniques are reevaluated in practice (cf. Table 4), they often produced only marginal or statistically insignificant improvements. This suggests that general-purpose fuzzing may have reached a point of diminishing returns: the low-hanging fruits—which often were *simple*, a property that has proven successful—to maximize the efficiency and effectiveness of coverage feedback have largely been explored.

Our findings do not suggest we, as the academic fuzzing community, should abandon AFL++ as a baseline. In fact, we argue the opposite is the case: Given its maturity, widespread adoption, and role as an integration hub, we believe that contributing improvements to AFL++ remains one of the most effective paths toward achieving impact in practice. At the same time, exploring orthogonal and specialized approaches is essential to overcome the observed plateau, even if such approaches do not directly integrate into AFL++. While specialized fuzzers can achieve superior performance in specific contexts, their domain assumptions often limit comparability and generalization. We thus consider both directions important and complementary.

We see this as a *call to action* for the fuzzing community. To push beyond the current plateau, future research may need to design more specialized or fundamentally novel approaches, rather than continuing to refine existing general-purpose fuzzers.

Finally, we should emphasize *reproducibility*, *upstream integrability*, and *practical relevance*.

References

[1] K. Popper, *The Logic of Scientific Discovery*. Routledge, 2005.

[2] P. McCulloch, D. G. Altman, W. B. Campbell, D. R. Flum, P. Glasziou, J. C. Marshall, and J. Nicholl, "No surgical innovation without evaluation: the IDEAL recommendations," *The Lancet*, vol. 374, no. 9695, 2009.

[3] J. P. Ioannidis, "Why Most Published Research Findings are False," *PLoS Medicine*, vol. 2, no. 8, 2005.

[4] T. S. Kuhn, *The Structure of Scientific Revolutions*. University of Chicago Press, 1970.

[5] C. Herley and P. Van Oorschot, "SoK: Science, Security and the Elusive Goal of Security as a Scientific Pursuit," in *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[6] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A Survey for Roadmap," *ACM Computing Surveys (CSUR)*, vol. 54, no. 11s, 2022.

[7] 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," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, 2019.

[8] Google, "OSS-Fuzz: Continuous Fuzzing for Open Source Software," https://github.com/google/oss-fuzz.

[9] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2024.

[10] M. Zalewski, "American Fuzzy Lop," http://lcamtuf.coredump.cx/afl/, accessed: May 17, 2026.

[11] M. Böhme, L. Szekeres, and J. Metzman, "On the Reliability of Coverage-Based Fuzzer Benchmarking," in *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2022.

[12] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," in *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[13] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "FuzzBench: An Open Fuzzer Benchmarking Platform and Service," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

[14] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining Incremental Steps of Fuzzing Research," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[15] A. Fioraldi, D. C. D'Elia, and D. Balzarotti, "The Use of Likely Invariants as Feedback for Fuzzers," in *USENIX Security Symposium*, 2021.

[16] X. Zhu and M. Böhme, "Regression Greybox Fuzzing," in *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[17] Z. Jiang, S. Gan, A. Herrera, F. Toffalini, L. Romerio, C. Tang, M. Egele, C. Zhang, and M. Payer, "Evocatio: Conjuring Bug Capabilities from a Single PoC," in *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[18] N. Bars, M. Schloegel, T. Scharnowski, N. Schiller, and T. Holz, "Fuzztruction: Using Fault Injection-based Fuzzing to Leverage Implicit Domain Knowledge," in *USENIX Security Symposium*, 2023.

[19] M. Lee, S. Cha, and H. Oh, "Learning Seed-Adaptive Mutation Strategies for Greybox Fuzzing," in *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2023.

[20] H. Zheng, J. Zhang, Y. Huang, Z. Ren, H. Wang, C. Cao, Y. Zhang, F. Toffalini, and M. Payer, "FISHFUZZ: Catch Deeper Bugs by Throwing Larger Nets," in *USENIX Security Symposium*, 2023.

[21] D. She, A. Storek, Y. Xie, S. Kweon, P. Srivastava, and S. Jana, "Fox: Coverage-guided Fuzzing as Online Stochastic Control," in *ACM Conference on Computer and Communications Security (CCS)*, 2024.

[22] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing as Markov Chain," in *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[23] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "MOPT: Optimized Mutation Scheduling for Fuzzers," in *USENIX Security Symposium*, 2019.

[24] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with Input-to-State Correspondence," in *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[25] kdsjZh, "Enhancement on Deterministic stage," https://github.com/AFLplusplus/AFLplusplus/pull/1972, accessed: May 17, 2026.

[26] AFLplusplus, "AFLplusplus," https://github.com/AFLplusplus/AFLplusplus, accessed: May 17, 2026.

[27] A. Fioraldi, A. Mantovani, D. Maier, and D. Balzarotti, "Dissecting American Fuzzy Lop: A Fuzzbench Evaluation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 32, no. 2, pp. 1–26, 2023.

[28] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types," in *USENIX Security Symposium*, 2021.

[29] lafintel, "Circumventing Fuzzing Roadblocks with Compiler Transformations," https://lafintel.wordpress.com/, accessed: May 17, 2026.

[30] Google, "American Fuzzy Lop," https://github.com/google/AFL, accessed: May 17, 2026.

[31] LLVM, "SanitizerCoverage," https://clang.llvm.org/docs/SanitizerCoverage.html, accessed: May 17, 2026.

[32] fuzzbench.com, "FuzzBench Reports," https://www.fuzzbench.com/reports/, accessed: May 17, 2026.

[33] vanhauser thc, "FuzzBench - Better Coverage Benchmarks," https://github.com/google/fuzzbench/issues/1794, accessed: May 17, 2026.

[34] A. Fioraldi, D. C. Maier, D. Zhang, and D. Balzarotti, "LibAFL: A Framework to Build Modular and Reusable Fuzzers," in *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[35] S. Groß, S. Koch, L. Bernhard, T. Holz, and M. Johns, "FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities," in *Symposium on Network and Distributed System Security (NDSS)*, 2023.

[36] G. P. Zero, "Fuzzilli," https://github.com/googleprojectzero/fuzzilli, accessed: May 17, 2026.

[37] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels," in *USENIX Security Symposium*, 2017.

[38] C. Lemieux and K. Sen, "FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage," in *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018.

[39] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *USENIX Security Symposium*, 2018.

[40] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a Desired Directed Grey-box Fuzzer," in *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[41] M. Cho, S. Kim, and T. Kwon, "Intriguer: Field-Level Constraint Solving for Hybrid Fuzzing," in *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[42] S. Nagy and M. Hicks, "Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[43] S. Nilizadeh, Y. Noller, and C. S. Pasareanu, "DifFuzz: Differential Fuzzing for Side-channel Analysis," in *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2019.

[44] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware Greybox Fuzzing," in *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2019.

[45] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim, "Fuzzing File Systems via Two-Dimensional Input Space Exploration," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[46] W. You, X. Liu, S. Ma, D. M. Perry, X. Zhang, and B. Liang, "SLF: Fuzzing without Valid Seed Inputs," in *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2019.

[47] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang, "ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[48] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring Deep State Spaces via Fuzzing," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[49] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, "SAVIOR: Towards Bug-Driven Hybrid Testing," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[50] S. Dinesh, N. Burow, D. Xu, and M. Payer, "RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[51] Y. Noller, C. S. Pasareanu, M. Böhme, Y. Sun, H. L. Nguyen, and L. Grunske, "HyDiff: Hybrid Differential Software Analysis," in *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2020.

[52] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su, "Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization," in *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[53] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu, "MemLock: Memory Usage Guided Fuzzing," in *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2020.

[54] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou, "EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit," in *USENIX Security Symposium*, 2020.

[55] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang, and Y. Liu, "MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs," in *USENIX Security Symposium*, 2020.

[56] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen, and Y. Sui, "Typestate-guided Fuzzer for Discovering Use-after-free Vulnerabilities," in *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2020.

[57] C. Zhou, M. Wang, J. Liang, Z. Liu, and Y. Jiang, "Zeror: Speed Up Fuzzing with Coverage-sensitive Tracing and Scheduling," in *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2020.

[58] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing," in *USENIX Security Symposium*, 2021.

[59] ——, "Same Coverage, Less Bloat: Accelerating Binary-only Fuzzing with Coverage-preserving Coverage-guided Tracing," in *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[60] G. Pan, X. Lin, X. Zhang, Y. Jia, S. Ji, C. Wu, X. Ying, J. Wang, and Y. Wu, "V-Shuttle: Scalable and Semantics-Aware Hypervisor Virtual Device Fuzzing," in *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[61] X. Ge, B. Niu, R. Brotzman, Y. Chen, H. Han, P. Godefroid, and W. Cui, "HyperFuzzer: An Efficient Hybrid Fuzzer for Virtual CPUs," in *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[62] G. Lee, W. Shim, and B. Lee, "Constraint-guided Directed Greybox Fuzzing," in *USENIX Security Symposium*, 2021.

[63] C. Salls, C. Jindal, J. Corina, C. Kruegel, and G. Vigna, "Token-Level Fuzzing," in *USENIX Security Symposium*, 2021.

[64] J. Wang, C. Song, and H. Yin, "Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing," in *Symposium on Network and Distributed System Security (NDSS)*, 2021.

[65] Z. Du, Y. Li, Y. Liu, and B. Mao, "Windranger: A Directed Greybox Fuzzer driven by Deviation Basic Blocks," in *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2022.

[66] C. Lyu, S. Ji, X. Zhang, H. Liang, B. Zhao, K. Lu, and R. Beyah, "EMS: History-Driven Mutation for Coverage-based Fuzzing," in *Symposium on Network and Distributed System Security (NDSS)*, 2022.

[67] R. Meng, Z. Dong, J. Li, I. Beschastnikh, and A. Roychoudhury, "Linear-time Temporal Logic guided Greybox Fuzzing," in *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2022.

[68] Y. Yu, X. Jia, Y. Liu, Y. Wang, Q. Sang, C. Zhang, and P. Su, "HTFuzz: Heap Operation Sequence Sensitive Fuzzing," in *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2022.

[69] P. Li, W. Meng, and K. Lu, "SEDiff: Scope-aware Differential Fuzzing to Test Internal Function Models in Symbolic Execution," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022.

[70] C. Myung, G. Lee, and B. Lee, "MundoFuzz: Hypervisor Fuzzing with Statistical Coverage Testing and Grammar Inference," in *USENIX Security Symposium*, 2022.

[71] G. Zhang, P. Wang, T. Yue, X. Kong, S. Huang, X. Zhou, and K. Lu, "MobFuzz: Adaptive Multi-objective Optimization in Gray-box Fuzzing," in *Symposium on Network and Distributed System Security (NDSS)*, 2022.

[72] J. Fu, J. Liang, Z. Wu, M. Wang, and Y. Jiang, "Griffin: Grammar-Free DBMS Fuzzing," in *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2022.

[73] P. Deng, Z. Yang, L. Zhang, G. Yang, W. Hong, Y. Zhang, and M. Yang, "NestFuzz: Enhancing Fuzzing with Comprehensive Understanding of Input Processing Logic," in *ACM Conference on Computer and Communications Security (CCS)*, 2023.

[74] P. Jauernig, D. Jakobovic, S. Picek, E. Stapf, and A.-R. Sadeghi, "DARWIN: Survival of the Fittest Fuzzing Mutators," in *Symposium on Network and Distributed System Security (NDSS)*, 2023.

[75] L. Jiang, H. Yuan, M. Wu, L. Zhang, and Y. Zhang, "Evaluating and Improving Hybrid Fuzzing," in *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2023.

[76] T. E. Kim, J. Choi, K. Heo, and S. K. Cha, "DAFL: Directed Grey-box Fuzzing guided by Data Dependency," in *USENIX Security Symposium*, 2023.

[77] D. Wang, Y. Li, Z. Zhang, and K. Chen, "CarpetFuzz: Automatic Program Option Constraint Extraction from Documentation for Fuzzing," in *USENIX Security Symposium*, 2023.

[78] Y. Wu, T. Zhang, C. Jung, and D. Lee, "DEVFUZZ: Automatic Device Model-Guided Device Driver Fuzzing," in *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[79] W. Li, J. Ruan, G. Yi, L. Cheng, X. Luo, and H. Cai, "PolyFuzz: Holistic Greybox Fuzzing of Multi-Language Systems," in *USENIX Security Symposium*, 2023.

[80] J. Yin, M. Li, Y. Li, Y. Yu, B. Lin, Y. zou, Y. Liu, W. Huo, and J. Xue, "RSFuzzer: Discovering Deep SMI Handler Vulnerabilities in UEFI Firmware with Hybrid Fuzzing," in *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[81] H. Huang, P. Yao, H. Chiu, Y. Guo, and C. Zhang, "Titan: Efficient Multi-target Directed Greybox Fuzzing," in *IEEE Symposium on Security and Privacy (S&P)*, 2024.

[82] Y. Zhang, Y. Liu, J. Xu, and Y. Wang, "Predecessor-aware Directed Greybox Fuzzing," in *IEEE Symposium on Security and Privacy (S&P)*, 2024.

[83] X. Liu, W. You, Y. Ye, Z. Zhang, J. Huang, and X. Zhang, "FuzzInMem: Fuzzing Programs via In-memory Structures," in *International Conference on Software Engineering (ICSE)*, 2024.

[84] P. Borrello, A. Fioraldi, D. C. D'Elia, D. Balzarotti, L. Querzoni, and C. Giuffrida, "Predictive Context-sensitive Fuzzing," in *Symposium on Network and Distributed System Security (NDSS)*, 2024.

[85] H. Rong, W. You, X. Wang, and T. Mao, "Toward Unbiased Multiple-Target Fuzzing with Path Diversity," in *USENIX Security Symposium*, 2024.

[86] Y. Yang, Y. Chen, R. Zhong, J. Chen, and W. Lee, "Towards Generic Database Management System Fuzzing," in *USENIX Security Symposium*, 2024.

[87] K. Zhang, X. Zhu, X. Xiao, M. Xue, C. Zhang, and S. Wen, "ShapFuzz: Efficient Fuzzing via Shapley-Guided Byte Selection," in *Symposium on Network and Distributed System Security (NDSS)*, 2024.

[88] K. Wang, M. Chen, L. He, P. Su, Y. Cai, J. Chen, B. Zhang, C. Feng, and C. Tang, "OSmart: Whitebox Program Option Fuzzing," in *ACM Conference on Computer and Communications Security (CCS)*, 2024.

[89] H. Zheng, F. Toffalini, M. Böhme, and M. Payer, "MendelFuzz: The Return of the Deterministic Stage," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2025.

[90] Z. Kong, S. Li, H. Huang, and Z. Su, "Sand: Decoupling Sanitization from Fuzzing for Low Overhead," in *International Conference on Software Engineering (ICSE)*, 2025.

[91] dkasak, "FairFuzz (afl-rb) Integration," https://github.com/AFLplusplus/AFLplusplus/issues/18, accessed: May 17, 2026.

[92] zhanggenex, "Incorporating EcoFuzz?" https://github.com/AFLplusplus/AFLplusplus/issues/380, accessed: May 17, 2026.

[93] virtuald, "Add manual mechanism to motivate AFL to pursue a particular input," https://github.com/AFLplusplus/AFLplusplus/issues/643, accessed: May 17, 2026.

[94] nbars, "Disabling PCGUARD edge pruning for AFL++ for evaluation," https://github.com/FOX-Fuzz/FOX/issues/3, accessed: May 17, 2026.

[95] C.-C. Hsu, C.-Y. Wu, H.-C. Hsiao, and S.-K. Huang, "Instrim: Lightweight Instrumentation for Coverage-guided Fuzzing," in *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, 2018.

[96] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, "Smart Greybox Fuzzing," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1980–1997, 2019.

[97] Quarkslab, "QBDI - A Dynamic Binary Instrumentation framework," https://github.com/QBDI/QBDI, accessed: May 17, 2026.

[98] G. P. Zero, "WinAFL - Fork of AFL for fuzzing Windows binaries," https://github.com/googleprojectzero/winafl, accessed: May 17, 2026.

[99] RICSecLab, "coresight-trace - ARM CoreSight-based tracing framework," https://github.com/RICSecLab/coresight-trace, accessed: May 17, 2026.

[100] Quarkslab, "TritonDSE - Dynamic Symbolic Execution framework," https://github.com/quarkslab/tritondse, accessed: May 17, 2026.

[101] L. Padgham, Y. Lee, S. Sadiq, M. Winikoff, A. Fekete, S. MacDonell, D. Kaafar, and S. Zollmann, "CORE Rankings." [Online]. Available: https://www.core.edu.au/conference-portal

[102] D. Maier, L. Seidel, and S. Park, "Basesafe: Baseband Sanitized Fuzzing through Emulation," in *ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2020.

[103] S. Schumilo, C. Aschermann, A. Jemmett, A. Abbasi, and T. Holz, "Nyx-net: Network Fuzzing with Incremental Snapshots," in *European Conference on Computer Systems (EuroSys)*, 2022.

[104] choller, "AFL - A Fork of AFL with Enhancements," https://github.com/choller/afl, accessed: May 17, 2026.

[105] Google, "Honggfuzz – Security Oriented, Feedback-driven Fuzzer," https://github.com/google/honggfuzz, accessed: May 17, 2026.

[106] akihe, "Radamsa – Fuzzer for Testing Software by Generating Inputs from Sample Data," https://gitlab.com/akihe/radamsa, accessed: May 17, 2026.

[107] LLVM Project, "LibFuzzer - A library for in-process, coverage-guided fuzzing," https://llvm.org/docs/LibFuzzer.html, accessed: May 17, 2026.

[108] N. Group, "TriforceAFL - AFL fork with QEMU full-system fuzzing support," https://github.com/nccgroup/TriforceAFL, accessed: May 17, 2026.

[109] Battelle, "afl-unicorn - AFL with Unicorn CPU emulator support," https://github.com/Battelle/afl-unicorn, accessed: May 17, 2026.

[110] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing New Operating Primitives to Improve Fuzzing Performance," in *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[111] P. Chen and H. Chen, "Angora: Efficient Fuzzing by Principled Search," in *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[112] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song, "Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing," in *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2019.

[113] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "NAUTILUS: Fishing for Deep Bugs with Grammars," in *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[114] M. Böhme, V. J. M. Manès, and S. K. Cha, "Boosting Fuzzer Efficiency: An Information Theoretic Perspective," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020.

[115] A. Fioraldi, D. C. D'Elia, and L. Querzoni, "Fuzzing Binaries for Memory Safety Errors with QASan," in *IEEE Secure Development (SecDev)*, 2020.

[116] S. Poeplau and A. Francillon, "Symbolic execution with SymCC: Don't interpret, compile!" in *USENIX Security Symposium*, 2020.

[117] ——, "SymQEMU: Compilation-based Symbolic Execution for Binaries," in *Symposium on Network and Distributed System Security (NDSS)*, 2021.

[118] P. Srivastava and M. Payer, "Gramatron: Effective Grammar-aware Fuzzing," in *International Symposium on Software Testing and Analysis (ISSTA)*, 2021.

Appendix A. Ethical Considerations

Our work explores fuzzing, a method for finding vulnerabilities in software. However, our goal is *not* to discover new software faults, but to systematically evaluate the current capabilities of fuzzing tools. To the best of our knowledge, no new bugs have been discovered throughout our re-evaluation on known benchmarks, so no coordinated disclosure was required. As all fuzzers, features, and targets considered in our paper are already open source, there are no concerns with releasing our evaluation-focused artifact.

Appendix B. Data Availability

To foster future research and enable reproducibility of our experiments, we release all code, including experimental setup, evaluation scripts, and targets used. We note that our work fundamentally builds upon already publicly available techniques and tools. That said, we publish our fuzzing environment, which can be used to rerun our experiments. Our artifact is available at https://doi.org/10.5281/zenodo.19887428.

Appendix C. Additional Results

In the following, we list additional results for interested readers.

All AFL++ versions on target BLOATY. Figure 13 shows all AFL++ versions used in their default setting for the target BLOATY.

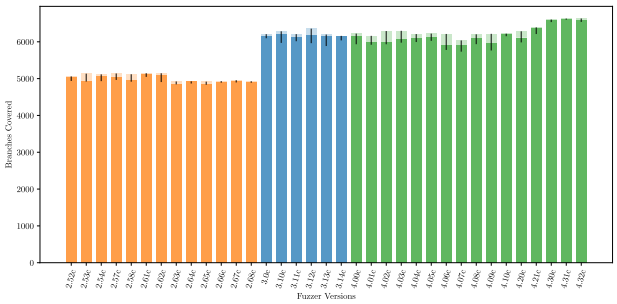


Figure 13: All AFL++ versions with default settings on the target BLOATY. We use different colors to visualize major versions.

Fuzzer Improvement with Outliers. Figure 14 shows the same data as Figures 3 and 4, except it also includes outlier values, which we omitted for readability in the main text. We observe these outliers as the fuzzer performs significantly differently on specific targets.

`CmpLog` Improvement between Versions 3.0c and 3.14c Figure 16 displays the improvement of the `CmpLog` feature over time, here exemplary visualized for FREETYPE2.

Case Study: Reduced Performance 3.0c – 4.10c. Across experiments, version *3.0c* consistently outperforms subsequent versions until *4.10c*, which eventually catches up in

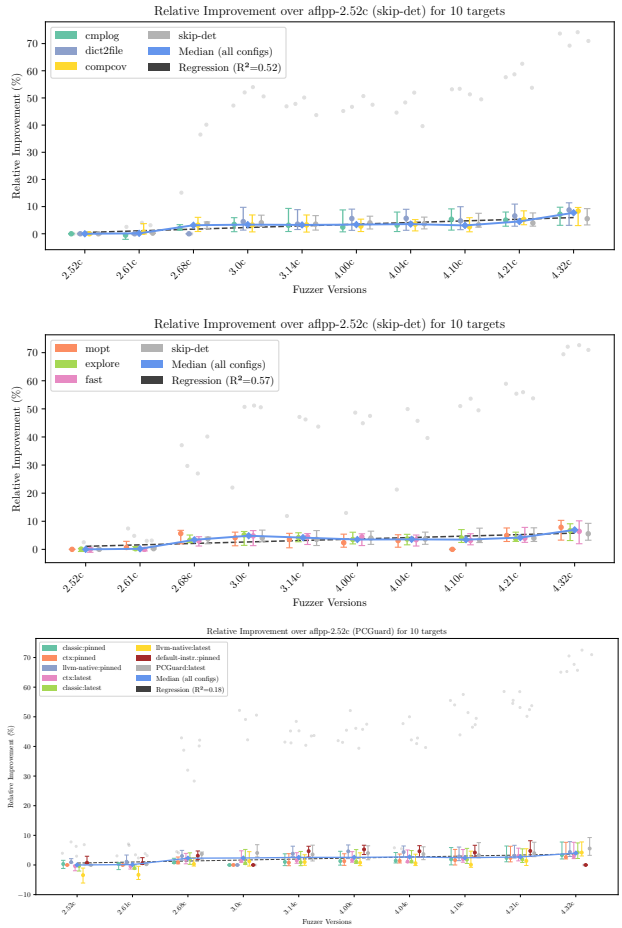


Figure 14: The plot of the different improvements with outliers. Grey markers indicate targets outside the 25th to 75th percentile.

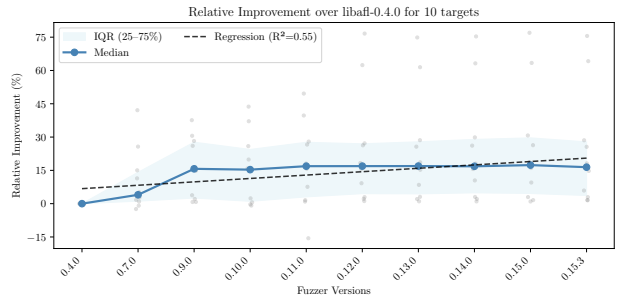


Figure 15: The plot shows the median relative improvement and regression per released LIBAFL version over 10 targets and 10 repetitions. The grey markers show the performance of each target per version.

coverage. These results raise the question of what changes led to the observed performance decline, especially since the most impactful configurations – such as the default seed schedule and the use of the deterministic stage – remain unchanged across these versions. Due to our selection of AFL++ versions, we skipped several releases between *3.0c* and *3.14c*, with the latter exhibiting noticeably worse performance. This decline is particularly evident when using the `dict2file` configuration across various targets and in baseline experiments where the deterministic stage is skipped for all versions or when utilizing *llvm-native* instrumentation. The consistent degradation observed in the versions following *3.0c* suggests a potential hidden regression affecting all subsequent versions. To investigate this issue, we conducted a targeted experiment using the `dict2file` feature on FREETYPE2, as this target exhibited the most pronounced performance drop. We aimed to pinpoint the changes responsible for the decline and assess their impact on the effectiveness of fuzzing. Comparing results across these versions reveals a mostly continuous decline in coverage with each new release (see Figure 17). Notably, the most significant performance drop occurs between *3.0c* and *3.10c*. To pinpoint the cause, we bisected the commits between these two versions, but the results were inconclusive. The performance degradation appears to stem from multiple incremental changes rather than a single, clearly identifiable breaking commit, making it difficult to determine the exact source of the regression.

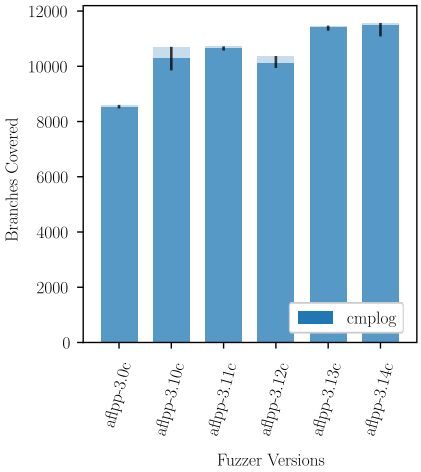


Figure 16: The performance improvement of `CmpLog` on the target FREETYPE2.

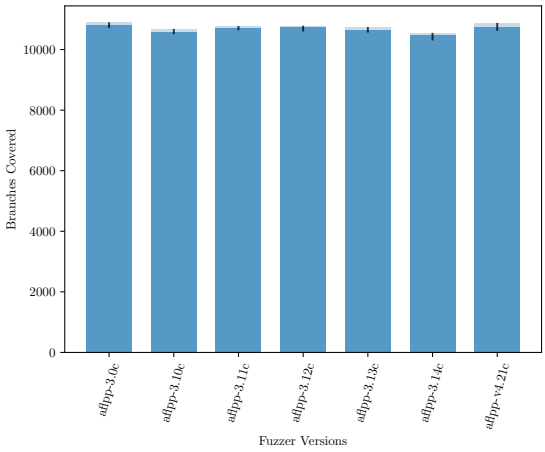


Figure 17: The performance decline on the target FREETYPE2 with the `dict2file`

Best Configuration Overview. Table 5 is complementary to Figure 8 and shows the best configuration per fuzzer and target.

TABLE 5: Best configurations per target and AFL++ version.

Target	2.52c	2.61c	2.68c	3.0c	3.14c	4.00c	4.04c	4.10c	4.21c	4.32c
curl	⊠	⊠	◆	⊕	★	⊕	⊕	⊕	⊕	⊕
harfbuzz	◆	▲	◆	●	⊠	⊠	⊕	★	▲	▲
libxslt	◆	◆	◆	⊕	⊕	⊕	⊕	⊕	⊕	⊕
openh264	●	▲	▲	◆	■	■	▲	●	◆	◆
openssl	★	★	◆	■	◆	■	■	■	■	●
freetype2	⊠	◆	◆	⊕	■	■	■	■	■	■
bloaty	◆	◆	◆	▲	▲	⊕	▲	▲	◆	◆
libjpeg	●	▲	▲	▲	▲	▲	▲	▲	▲	▲
libpcap	◆	◆	◆	▲	▲	▲	▲	●	▲	▲
stb	★	▲	◆	⊠	■	■	■	■	⊕	◆

explore: ⊠ fast: ● cmplog: ■ mopt: ◆
skip-det: ◆ deterministic: ★ compcov: ▲ dict2file: ⊕

Integrated Works in AFL++.

We list features of AFL++ (beyond the scope of top-tier venues) in Table 6. Notably, many features originate from non-academic contexts or non-prestigious venues.

TABLE 6: Overview of fuzzing tools and techniques integrated (or that inspired features) in AFL++ sourced from the original AFL++ paper and public sources. Entries show only major works; numerous other improvements (e.g., [102], [103]) have been made to already-existing features or to address bugs but are omitted here. The color coding categorizes contributions as special usecase, feature, or improvement.

Tool	Year	Published	Based On	Integrated component	
Holler's AFL Fork [104]	2015	–	AFL	custom mutators, file instrument list	*improvement*
Honggfuzz Mutators [105]	2015	–	honggfuzz	honggfuzz mutators	*feature*
AFLFast [22]	2016	CCS	AFL	AFL++ power schedules	*improvement*
LAF-Intel / CompCov [29]	2016	–	AFL	compare splitting	*feature*
Radamsa [106]	2016	–	–	mutator	*feature*
WINAFL [98]	2016	–	AFL	inspiration for QEMU persistent mode	*special usecase*
LibFuzzer [107]	2016	–	AFL	custom mutator	*special usecase*
TriforceAFL [108]	2017	–	–	forkserver patch for afl-tmin	*improvement*
AFLUnicorn [109]	2017	–	AFL	Unicorn mode	*special usecase*
QuarksLab QBDI [97]	2017	–	–	example for harnessing with QBDI	*special usecase*
Perffuzz [110]	2017	CCS	AFL	LKM inspired by Perffuzz / snapshot	*special usecase*
New Operating Primitive [110]	2017	CCS	AFL	snapshotting kernel module (deprecated)	*improvement*
InsTrim [95]	2018	BAR Workshop	AFL	instrumentation	*feature*
FairFuzz [38]	2018	ASE	AFL	inspired the rare edge scheduler	*feature*
Angora [111]	2018	S&P	–	context-sensitive coverage	*feature*
MOpt [23]	2019	USENIX	AFL	mutation scheduling	*feature*
RedQueen [24]	2019	NDSS	kAFL	CmpLog, input-to-state mutator	*feature*
Ngram & CTX [112]	2019	RAID	–	ngram-coverage and context sensitive feedback	*feature*
AFLSmart [96]	2019	TSE	AFL	high order structural mutations	*improvement*
Untracer [42]	2019	S&P	AFL	fast fuzzing of binary only libraries	*special usecase*
Nautilus [113]	2019	NDSS	–	grammar fuzzer	*special usecase*
Entropic [114]	2020	FSE	libfuzzer	entropy based scheduler, rare seeds	*feature*
QASAN [115]	2020	SecDev	AFL	QEMU mode ASAN implementation	*special usecase*
SymCC [116]	2020	USENIX	–	as custom mutator	*special usecase*
SymQEMU [117]	2021	NDSS	–	as custom mutator	*special usecase*
Nyx [28]	2021	USENIX	–	full system emulation and snapshotting	*special usecase*
Gramatron [118]	2021	ISSTA	–	custom grammar mutator	*special usecase*
RICSecLab coresight mode [99]	2022	–	–	binary-only fuzzing for ARM64	*special usecase*
QuarksLab TritonDSE [100]	2023	–	–	custom mutator	*special usecase*
MendelFuzz [89]	2025	FSE	AFL++	fast deterministic stage	*improvement*
SAND [90]	2025	ICSE	AFL++	reduced sanitizer overhead	*feature*