StorFuzz: Using Data Diversity to Overcome Fuzzing Plateaus

Leon Weiß
Ruhr University Bochum
Bochum, Germany
leon.weiss@rub.de

Tobias Holl
Ruhr University Bochum
Bochum, Germany
tobias.holl@rub.de

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

Abstract

Fuzzing is widely used to discover software bugs and vulnerabilities. Unfortunately, real-world long-running fuzzing campaigns often plateau and no progress can be made anymore, leaving code areas untested. State-of-the-art fuzzers leverage code coverage to measure progress and reach new areas, but this is insufficient to capture all program behavior, as code coverage may be the same for different behaviors, thus preventing progress and masking bugs.

In this paper, we introduce StorFuzz, a novel technique to overcome fuzzing plateaus and improve on code coverage by leveraging our new data coverage. StorFuzz automatically identifies and instruments *memory stores* to capture changes in program behavior invisible to control flow, which it uses to diversify the saturated corpora of plateaued campaigns. StorFuzz leverages this diversified corpus of test cases that changed internal states to improve navigation of the input space, which also enables conventional fuzzers to improve their code coverage. We implement StorFuzz in LibAFL and evaluate on FuzzBench, starting from a corpus that is saturated by multi-month OSS-Fuzz fuzzing campaigns and LibAFL.

We show that StorFuzz successfully generates new coverage for plateauing campaigns of widely-used and well-fuzzed software, leading to the discovery of 50 new bugs in 7 OSS-Fuzz projects, like VLC and PHP, with some bugs having been present in the code for *14 years*. Our approach significantly outperforms both the state-of-the-art fuzzer LibAFL and data-guided fuzzer DDFuzz in 11 of 23 FuzzBench benchmarks, while performing equally on all others. StorFuzz is also complementary to WingFuzz, an approach guided by static data, as both fuzzers cover distinct code regions.

Source Code and Data: `github.com/rub-softsec/StorFuzz`

CCS Concepts

• **Security and privacy → Software security engineering**; **Systems security**; • **Software and its engineering → Software testing and debugging**.

Keywords

Fuzzing, Coverage Plateaus, Vulnerability Discovery

ACM Reference Format:
Leon Weiß, Tobias Holl, and Kevin Borgolte. 2026. StorFuzz: Using Data Diversity to Overcome Fuzzing Plateaus. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773179

ICSE '26, Rio de Janeiro, Brazil
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2025-3/2026/04
https://doi.org/10.1145/3744916.3773179

1 Introduction

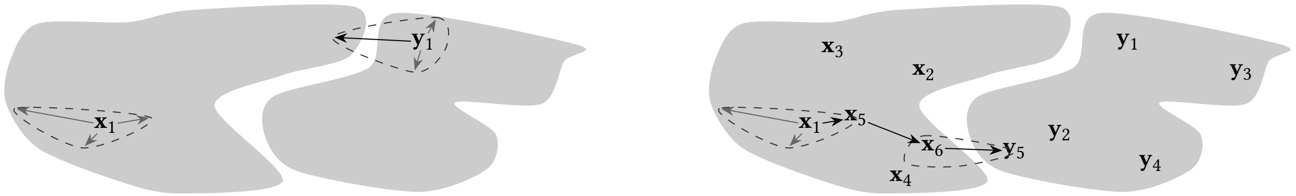
Fuzzing has proven to be a successful technique in finding bugs by supplying random inputs to its target and observing whether it behaves irregularly. However, after fuzzing for some time, code coverage plateaus and there is little to no further advancement. In most cases, this does not mean that the fuzzer has covered all reachable code, but rather that it has trouble finding an input that progresses to a new code area. Thus, the remaining areas continue to be untested and bugs stay undiscovered. In fact, the long-running campaigns of OSS-Fuzz only cover 39% of all lines in the median C/C++ project [10], leaving most code untested. Moreover, even in tested areas, they failed to discover latent bugs that are at least 14 years old, and despite the fact that they fuzzed the respective project (VLC) in OSS-Fuzz for almost 4 years (see Section 4.10.2).

Since the inception of fuzzing, numerous researchers have published on building [8, 9, 15] and evaluating [12, 23, 28] fuzzers. Almost all of them focused on fuzzing for a short 24–48 hours or tried to reduce the fuzzing time even more [16, 20]. In practice, however, real-world fuzzing campaigns are long-running to basically permanent. For example, Google's OSS-Fuzz [1] initiative has demonstrated that fuzzing programs for a long time is beneficial. Its track record has shown that even after a long time there are still new bugs being found when new areas are covered.

Modern coverage-guided fuzzers [8, 9, 15, 19] focus on code coverage and only keep inputs when they *immediately* lead to new code coverage. While this efficiently leads to progress during the initial fuzzing phase, it leads to a stalled corpus when code coverage plateaus. In this paper, we introduce StorFuzz to escape these plateaus by carefully increasing input diversity. We also show that our approach improves code coverage of state-of-the-art fuzzers after long fuzzing periods. StorFuzz achieves this by combining code coverage with information on stored values.

In this paper, we make the following contributions:

- We introduce StorFuzz, a novel fuzzing approach dedicated to overcoming the coverage plateau by taking stored values into account when retaining test cases, ultimately improving on the code coverage of modern fuzzers that have plateaued.
- We show that StorFuzz improves code coverage after other fuzzers have plateaued. Leveraging our generic approach to introduce diversity into the fuzzing corpus, we ensure independence of the plateaued fuzzer.
- We demonstrate StorFuzz's bug-finding abilities by discovering 50 new, previously unknown bugs and vulnerabilities in 7 heavily-fuzzed OSS-Fuzz projects, such as VLC and PHP.
- We make StorFuzz and our artifacts openly available to allow others to replicate and build upon it:
`https://github.com/rub-softsec/StorFuzz`



(a) Using code coverage alone, we only keep one input per equivalence class. Fuzzing plateaus when the current corpus entries are placed poorly in their equivalence classes or require a large, improbable mutation step to reach new code.

(b) Using additional data coverage, we keep multiple inputs for each equivalence class. This allows us to reach the next code coverage equivalence class via incremental mutation steps inside the same class (e.g., $\mathbf{x}_1 \to \mathbf{x}_5 \to \mathbf{x}_6 \to \mathbf{y}_5$).

Figure 1: Program input space. Shaded areas are sets of inputs with the same code coverage (code coverage equivalence classes).

1.1 Motivation

From a theoretical standpoint, we can think of mutational fuzzing as (1) sampling from the target program's input space by choosing a known input and mutating it to advance to a new element in the input space. Whether the fuzzer keeps this mutated input is then (2) decided based on some retention criteria. Conventional state-of-the-art code coverage-guided fuzzing checks if it leads to new code coverage. This process effectively partitions the input space into classes of inputs that are equivalent in their code coverage.

Correspondingly, we can model fuzzing as repeated attempts to find new inputs in unknown code coverage equivalence classes by mutating known inputs. In Figure 1, the shaded areas illustrate such equivalence classes. Any point inside an area is an input that results in the same code coverage. The fuzzer now samples a new input by selecting a known input (e.g., $\mathbf{x}_1$) and mutating it using one of its mutation operations (illustrated as arrows). This leads to a new input whose location in the input space is constrained by the selected input and the mutation operations (dashed line).

Typical code coverage-guided fuzzers retain only one element for each class and discard any subsequent test cases that fall into the same class. While this ensures efficient progress in poorly explored programs, it limits the fuzzers to reaching other equivalence classes by the distance of a retained element to a new neighboring equivalence class and the available mutation operations. For example, in Figure 1a, the fuzzer can advance from test case $\mathbf{y}_1$ to a new test case in the equivalence class $\mathbf{x}$. However, starting from $\mathbf{x}_1$, it is impossible to reach $\mathbf{y}$ using the available mutation operations.

Different retention criteria (with a more sensitive metric) allow keeping multiple inputs per class, shown in Figure 1b. These additional test cases in each code coverage equivalence class create the opportunity for simple "short" mutations to reach the next equivalence class over the course of multiple chained mutations. In this case, starting from test case $\mathbf{x}_1$, the fuzzer can now keep $\mathbf{x}_5$ even though it does not immediately result in new code coverage. Ultimately, this allows reaching the new code coverage equivalence class $\mathbf{y}$ via multiple intermediate steps ($\mathbf{x}_1 \to \mathbf{x}_5 \to \mathbf{x}_6 \to \mathbf{y}_5$).

In the first hours of a fuzzing campaign, modern code coverage-guided fuzzers, like LibAFL, can efficiently discover new code coverage equivalence classes. By only keeping a single test case per discovered equivalence class, they focus on the discovery of new code coverage instead of thorough exploration of known equivalence classes and they ignore different states encoded in data. After some time, in long-running campaigns, fuzzers reach a point where they cannot discover new code because of poor placement of the retained test cases in their equivalence classes.

When code coverage alone plateaus, we improve diversity inside these code coverage equivalence classes using StorFuzz. We build on the observation that virtually all programs contain some notion of state, where certain decisions, deductions, or results are *stored* that influence execution in other program parts. StorFuzz combines stored values with code coverage, keeping more test cases that cover the same code but which result in different internal states as defined by data. These test cases are distributed around the equivalence class and the improved variety ultimately creates new opportunities for fuzzers to reach new code.

2 Background

2.1 Coverage-Guided Fuzzing

Coverage-guided fuzzers continually try to improve on their *coverage metric* by generating new test cases (e.g., via mutating known inputs) and supplying them to the target. They then monitor if the new test cases improve on their metric and add any test cases to the corpus for which they observe new coverage.

Coverage-guided fuzzers most often use some notion of code coverage as their guiding metric, as it is also the most used measure of evaluating software testing efficacy. Different granularities of code coverage exist, ranging from the most precise path coverage to the least precise block or function coverage. Collecting more detailed coverage data, like path coverage, can provide more fine-grained feedback to the fuzzer, but it also introduces a performance penalty due to increased overhead. Today, most modern fuzzers use branch/edge coverage [8, 9, 15, 19], as this strikes the best balance between accuracy and performance overhead.

2.2 Limitations of Code Coverage

Code coverage metrics are, by design, limited to measuring control flow changes, which is only part of a program's execution. With edge coverage, they cannot detect differences between all control flow paths, and much less in data that the program processes. This is a significant limitation, as the state of a program's execution does not only depend on the code paths taken but also on data. Ignoring data when considering the program's state can lead to an incomplete understanding of program behavior and miss important aspects that could uncover new bugs or vulnerabilities.

For example, it is common to split file processing into file format detection and various parsing stages. Information obtained in one stage (e.g., file format, version, or encoding details) is written to data structures that are passed to subsequent phases. A fuzzer considering only code coverage will discard inputs with different format characteristics when they hit a common error case before reaching the processing stage where this information influences control flow. On the other hand, fuzzers with more fine-grained coverage can detect the state change before the control flow changes and iteratively derive valid files with different format characteristics to uncover bugs in other parts of the code. We explore such a bug (vlc-6, found by StorFuzz in the VLC media player) in Section 4.10.

Fundamentally, while code coverage is a powerful tool for guiding fuzzers, and often seen as a panacea, it fails to capture important differences in program behavior that are driven by data.

2.3 Data Coverage Metrics

Incorporating data into the coverage metrics is a natural extension, as it allows capturing a more complete view of a program's execution. In fact, prior work has explored including data at different precision, for example, by inspecting def-use-chains [14, 22], or by tracking individual input bytes to select mutations [11, 26].

More selective approaches try to locate state variables in specific kinds of programs, like network protocol implementations [4, 24] or the Linux kernel [25, 40], which limits them to their specific program types. Others require developer annotations [2].

Concrete values frequently influence control flow in comparisons. For example, AFL++ includes an approach to split up large comparisons into smaller ones to create additional branches, which are more amenable to conventional code coverage [17]. Other approaches, like cmplog [3], signal the comparison values to the fuzzer, or track partially fulfilled comparisons [19]. As one operand in comparisons is often a static value and known at compile time, dictionaries with their values can be extracted at compile time. Wang et al. [33] extend this idea by aiming to detect them at runtime.

Incorporating data into coverage metrics provides a more fine-grained view of the program's execution. Effectively, it adds a new dimension that partitions the code coverage equivalence classes, subdividing them into multiple smaller ones. In turn, this allows distinguishing between program states that are identical in code coverage but actually represent different program behavior.

State-of-the-art approaches like DDFuzz [22] and WingFuzz [33] require the data to be used (e.g., in comparisons) for them to be able to detect the state changes, often approximating code coverage. StorFuzz, on the other hand, discovers new states as they are reached and before they influence control flow in any way, making StorFuzz more sensitive to state changes and allowing for incremental steps toward new code coverage.

3 StorFuzz

StorFuzz is a novel fuzzing approach that can achieve new code coverage where other fuzzers have plateaued and transfers these gains back to the previously plateaued fuzzer. It achieves this progress by complementing code coverage with a novel light-weight data coverage tracking *stored values* to capture state and introduce diversity to the code coverage equivalence classes.

```c
uint8_t value_reduction(uint64_t v) {
  return (v & 0xFF) ^ ((v & 0xFF00) >> 8)
}
```

Listing 1: StorFuzz's value reduction function.

At a high level, we track data coverage by combining the stored value with the location of the store instruction (see Section 3.1). To obtain this information, we instrument the target using a custom LLVM compiler pass, leveraging static analysis to minimize the overhead of our instrumentation. We inject our instrumentation in the form of small code snippets that expose data coverage information via a bitmap. To improve performance, we instrument only those stores that are likely to affect state (see Section 3.2). Our custom LibAFL-based in-process fuzzer evaluates the data coverage bitmap as well as the code coverage information generated by LibAFL's edge coverage pass (see Section 3.3). StorFuzz then decides whether to keep test cases based on a combination of data and code coverage, allowing for a more comprehensive exploration of the program's behavior.

3.1 Collecting Data Coverage

The main objective of StorFuzz is to observe the diversity of values stored in a variable. The key idea of our coverage approach is that we do not need to know whether two program statements store to the same source-level variable. It is sufficient to detect when the same statement writes a value to memory that we have not seen it write in any prior execution. We leverage analysis information from the compiler to optimize the locations that StorFuzz instruments.

First, we generate a unique ID for each statement that we want to instrument. At runtime, our instrumentation combines this ID with the value being stored, creating a unique identifier for each stored value at this instruction. We use the combination of the ID and the stored value as an index into our data coverage bitmap, where we set the corresponding bit to 1. This allows us to keep track of the different values stored by each statement.

A naïve approach that directly uses the observed values, which are up to 64 bits in size, would require a prohibitively large coverage map. Considering that the median FuzzBench benchmark program has 2,485 stores that we need to instrument (see Section 3.2), we would need a map of 5,730 exabytes ($2485 \cdot 2^{64}$ entries) to ensure a collision-free mapping, which is far too large to accommodate.

Instead, we introduce two optimizations: (1) *Value reduction*, reducing all values to 8 bits before combining them with the unique ID and (2) *bit compression*, using each byte in the coverage map for 8 different ID/value combinations.

For value reduction, we employ a computationally inexpensive approach that truncates/zero-extends the stored value to 16 bits and then combines its upper and lower half using XOR. Listing 1 shows the reduction in C, though it is actually implemented in LLVM IR (Intermediate Representation). With this reduction approach, we introduce collisions for different values stored by the *same* statement. This allows us to control the coverage granularity by accounting for at most 256 distinct values per store instruction. We selectively instrument stores to minimize the instrumented stores that do not contribute to state (e.g., we do not instrument copy operations of input bytes, see Section 3.2). In cases where we fail to exclude them, our value reduction acts as a secondary filter step, as most of these instructions quickly saturate the 256 discernible values without adding too much noise to the corpus.

For bit compression, we generate our unique IDs as tuples of an offset in the bitmap and an 8-bit bit-mask, effectively increasing the set of possible IDs by a factor of 8 while retaining the bitmap’s size. This does not increase the computational complexity at runtime because we use an OR operation to store the observation in our bitmap, as we do not need to track the frequency a certain value has been observed. By default, we use a data coverage map with size 2^{17} = 131,072 bytes, which allows us to instrument 4,096 store statements without collisions between locations. This is more than the median number of 2,485 instrumented stores across the 23 FuzzBench benchmarks. For 9 benchmarks, we instrument more than 4,096 stores, but we use the same map size for consistency, accepting potential collisions between locations.

3.2 Instrumentation

We instrument the target during compilation using our custom LLVM compiler pass. We implemented it in such a way that it is easy to integrate with LibAFL and remains compatible with the state-of-the-art code coverage pass. We use clang/LLVM to inject our instrumentation directly into the LLVM IR and synthesize it in a later stage. This allows us to seamlessly integrate with the compilation process of our targets without requiring modifications to the build system and ensures future extensibility.

Contrary to works like WingFuzz, lafintel, or REDQUEEN, which focus on loads or comparisons [3, 17, 33], our novel data coverage introspects operations that *change* state and we instrument stores. Hence, we do not need to identify any comparisons to the value, enabling us to capture data coverage earlier and even in places where other approaches are limited.

We instrument store locations based on the following criteria:

- The store must target a memory location and not a register, as these are changed very frequently to hold intermediate results during computation and therefore are not suitable to retain state.
- The store must not target a local variable on the stack (**`alloca`**), because they are unlikely to affect broader (program) state.
- The stored value must be unknown at compile time (i.e., it must not be a constant). Instrumenting constant stores duplicates code coverage, as the same value will be stored every time.
- The store target must not be a loop counter. These are implicitly tracked by code coverage through the counters of the loop’s edges, so instrumenting them would be redundant.
- The store must not be a mem-to-mem copy, such as a value that immediately comes from a **`load`** instruction. Such values must have been written to memory before, and we have already recorded them in our instrumentation of the initial write.
- The original type, before any potential casts, of the stored value must not be a floating point number or of a multi-element type (i.e., struct, vector, or array). Crucially, we instrument the more fine-grained stores in individual elements of vectors/arrays or members of structs, and we only exclude stores of the potentially large values as a whole. We also explicitly exclude pointers, as they depend on non-determinism, like ASLR and allocators, while adding little to no insight into the program state.

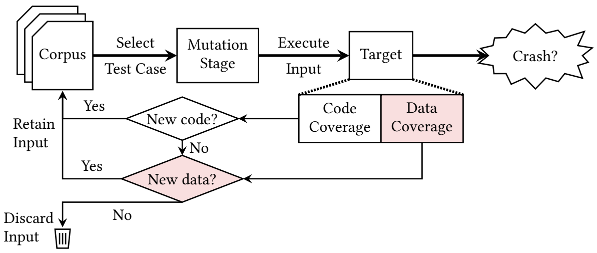


Figure 2: StorFuzz records the novel data coverage in a separate map and keeps non-crashing inputs if code or data feedback report new coverage.

We focus our instrumentation on integer-like values, because these are mainly responsible for describing program state.

We further limit our instrumentation to basic blocks (BBs) with at most 9 stores. We manually investigated the number of stores per BB in relation to the implemented functionality and found that BBs with many stores often do not change relevant state, but implement functionality like elliptic curve arithmetic (e.g., in openssl).

3.3 Fuzzing

We implemented StorFuzz as an in-process fuzzer using the popular fuzzing framework LibAFL. We chose LibAFL to leverage existing infrastructure and allow others to integrate our components into their own fuzzers. The concept behind StorFuzz is *independent* of LibAFL and can readily be used with other fuzzers or frameworks. Figure 2 shows StorFuzz’s fuzzing pipeline.

Beyond edge coverage, StorFuzz includes a customized MapFeedback component that is responsible for interpreting each bit in the data coverage map recorded by our instrumentation as its own data coverage entry (see Section 3.1). Each bit in the bitmap corresponds to a unique combination of a store instruction’s ID and the value being stored. When a store instruction is executed with a new value, the corresponding bit in the bitmap is set to 1, indicating that a new data coverage point has been reached.

To decide whether a newly generated test case is interesting, StorFuzz considers not only code coverage but also accounts for data coverage. StorFuzz retains all test cases that result in the discovery of a new code coverage equivalence class and those that further explore a known code coverage equivalence class by encountering different data coverage states.

In other words, if a test case leads to executing a new edge (determined by edge coverage) or storing a new value at a store instruction (determined by data coverage), StorFuzz marks it as interesting and adds it to the queue for further mutation and exploration. StorFuzz’s fuzzing loop then follows the proven approach of scheduling a queue entry for mutation, mutating, and feeding it to the target, which updates the coverage information.

StorFuzz improves data diversity in the code coverage equivalence classes by leveraging our novel data coverage. Instead of being limited to a single test case per class, StorFuzz can use multiple inputs to move through the class incrementally in search of new code coverage (see Section 1.1, $\mathbf{x}_1\rightarrow\mathbf{x}_5\rightarrow\mathbf{x}_6\rightarrow\mathbf{y}_5$), enabling it to overcome coverage plateaus.

4 Evaluation

We evaluate the efficiency and efficacy of StorFuzz by investigating the following research questions:

RQ 1: Can StorFuzz escape fuzzing plateaus?

RQ 2: How much diversity does our instrumentation introduce?

RQ 3: Do potential improvements of StorFuzz transfer to other fuzzers in subsequent campaigns?

For each research question, we investigate both whether StorFuzz can outperform the baseline code coverage-guided fuzzer and a state-of-the-art fuzzer with a more fine-grained metric.

In addition, we investigate potential performance overhead (see Section 4.8) and conduct an ablation study of our individual design choices (see Section 4.9). We also fuzz current versions of well-fuzzed OSS projects as case studies to demonstrate that StorFuzz is capable of finding new bugs (see Section 4.10).

We assess both the impact of our approach and its potential for enhancing the discovery of new program behaviors or bugs by measuring coverage gains and comparing them across different fuzzers and benchmarks. To ensure fair comparisons, we evaluate code coverage only as measured by the state-of-the-art fuzzing benchmarking suite FuzzBench and we do not consider the coverage metrics reported by the individual fuzzers.

As StorFuzz is built on the LibAFL framework, we compare our fuzzer to the LibAFL "FuzzBench fuzzer" [39]. With LibAFL as our baseline, we eliminate other factors (e.g., changes in mutation operations or slightly different code coverage instrumentation) that may lead to higher code coverage and we accurately focus our evaluation on the core contribution of StorFuzz, our novel data coverage. We slightly extended the baseline fuzzer to obtain run-time statistics. Our code is available as open source [35].

We also compare StorFuzz against DDFuzz [22], which includes more fine-grained coverage considering data-flow. We chose its LibAFL implementation [37], to rule out changes due to a different base fuzzer. In an earlier experiment, we also considered ngram [32, 38] and DatAFLow [14], but found DDFuzz to perform better.

We later also compare to WingFuzz [33] (see Section 4.7), which is not guided by data-flow information but by usage of static data. We do this in a separate evaluation because WingFuzz is not based on LibAFL but on libFuzzer. For WingFuzz we can only evaluate on **RQ 3** (and to a lesser extent on **RQ 1**) because libFuzzer, as the underlying infrastructure, reloads the corpus on every crash and therefore does not produce reliable data. Additionally, the impact of libFuzzer's mutation operations on the results is unclear, which makes fairly comparing the coverage metrics impossible without re-implementing them for LibAFL. We also tested SGFuzz [4] as another libFuzzer-based approach for stateful programs, but found that it performs worse than WingFuzz and we omit it for clarity.

4.1 Experiment Design

StorFuzz aims to escape from code coverage plateaus of long-running fuzzing campaigns. This is an entirely different problem than what prior work tries to solve, which aims to achieve higher code coverage in the first 24–48 hours but does not investigate long-term performance. Prior work fuzzers might therefore perform worse in code coverage than LibAFL for real-world fuzzing campaigns but better in the first hours, which they have been optimized for. Thus, our experimental design differs from prior work.

Our experiments are organized as follows:

Getting to the Plateau. In order to test whether StorFuzz can escape a plateau, we first need to obtain a saturated corpus. We take public OSS-Fuzz corpora for each benchmark and extend them by running a state-of-the-art fuzzer for an extended period of time. The resulting corpora are well saturated. We verify saturation by measuring code coverage over time for the saturating fuzzing runs.

Escaping the Plateau. To answer **RQ 1**, we compare by how much StorFuzz and two state-of-the-art fuzzers improve code coverage over the saturated corpora in 24 hours. Specifically, we compare StorFuzz to LibAFL as a purely code coverage-guided fuzzer and DDFuzz as a fuzzer with a more fine-grained coverage metric.

Assessing Diversity. We answer **RQ 2** by investigating the metrics' sensitivities. We build upon an observation by Wang et al. [32], to measure the sensitivity of a metric by assessing corpus growth compared to the baseline. Therefore, we compare the corpus sizes after 24 hours of fuzzing with StorFuzz, LibAFL, and DDFuzz.

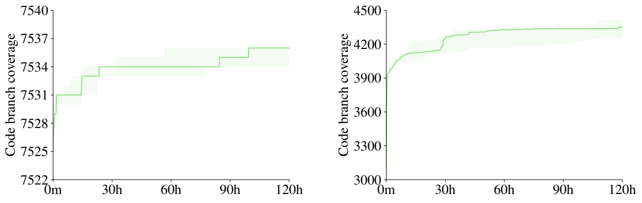
Transferring Diversity. To answer **RQ 3**, we need to compare the code coverage increase achieved by switching between StorFuzz and LibAFL or continuously running LibAFL. As prior work [27] has shown that restarting a fuzzer is beneficial to its code coverage, we also compare against a version that restarts LibAFL at the same interval. Additionally, we compare against the state-of-the-art data coverage-guided fuzzer WingFuzz [33].

4.2 Experiment Environment

We conduct all our coverage experiments on two servers with identical software and hardware configuration. Both are running Debian 12.8 with Docker 27.3.1. Each has 2,048 GB of DDR5 RAM and two AMD EPYC 9754 CPUs with 128 physical cores running at 2.25 GHz each, that is, 256 physical cores total per server. We disable hyperthreading (no logical cores), Turbo Boost Mode, and dynamic frequency scaling for the CPUs to guarantee that each fuzzer has exclusive access to a dedicated physical core and that the CPU frequency remains consistent throughout our experiments.

We pin each fuzzing instance to one dedicated physical core, preventing them from competing for CPU resources. Moreover, per server, we utilize only 230 cores for fuzzing and 15 for coverage measurements, leaving 11 cores idle, to avoid performance degradation due to system or FuzzBench orchestration needs.

We set up our experimental environment based on FuzzBench [23] and conducted the experiments following best practices [28], to provide a fair and accurate comparison of the different fuzzers. With FuzzBench, we evaluate code coverage using llvm-cov on dedicated coverage binaries. Our coverage plots include 95% confidence intervals using bootstrap. We repeated coverage experiments 10 times to account for randomness. Our evaluation is the result of fuzzing for more than 15 CPU-years.



(a) bloaty

(b) openthread

Figure 3: Branch coverage for LibAFL (●) starting from the OSS-Fuzz corpus. While coverage does not plateau for openthread, its growth still drops significantly. We plot median and 95% confidence intervals over 5 trials.

4.3 Getting to the Plateau

We obtain the saturated and plateauing corpus for each benchmark through the following approach:

(1) *We start from the OSS-Fuzz corpus.* This corpus is derived by continuously fuzzing the target programs with AFL++, honggfuzz, and libFuzzer. It is already saturated. It serves as the starting point for our experiments.

(2) *We run 5 LibAFL fuzzers individually for 120 hours*, each starting from the OSS-Fuzz corpus. Each instance runs independently for 5 days, allowing it to explore the target program's input space and generate new inputs for an extended time. By running multiple instances, we take advantage of fuzzing's inherent randomness and cover a wider range of program behaviors.

(3) *We merge the resulting queues* of all five 120-hours fuzzing campaigns and remove duplicate inputs based on their SHA-256 hash, achieving a more thoroughly saturated corpus.

The resulting corpus is well-saturated and the shared state of a saturated OSS-Fuzz corpus and multiple long-running LibAFL campaigns. It is our starting point for evaluating StorFuzz's effectiveness in escaping fuzzing plateaus. The initial OSS-Fuzz corpora and the resulting saturated corpora are available as open data [34].

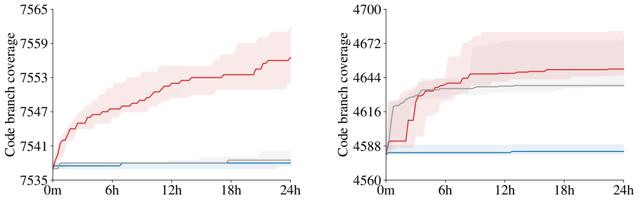
4.3.1 Results. Overall, our results show a clear decrease in coverage growth over time for all benchmarks, indicating saturation. Most benchmarks (17/23) clearly plateaued, like bloaty in Fig. 3a. Others (6) report small coverage improvements up until the end of the 120h window, like openthread in Fig. 3b, indicating that they have not completely plateaued yet, though they are about to.

4.4 Escaping the Plateau

To investigate **RQ 1**, we run StorFuzz and the state-of-the-art fuzzers LibAFL and DDFuzz on the saturated corpus for 24 hours. We compare code coverage gains of StorFuzz against those of LibAFL to determine if our approach can discover new program behavior and increase coverage over code coverage-guided fuzzers. To investigate if our novel data coverage improves over existing fine-grained coverage techniques, we also compare to DDFuzz.

Table 1: Diversifying the *saturated* corpus with StorFuzz performs equal to or better than both LibAFL and DDFuzz on all 23 benchmarks. We report median branch coverage of 10 24h-trials and difference to the seed set's coverage, the best fuzzer is highlighted. We omit benchmarks for which no fuzzer improves. Where StorFuzz improves over LibAFL the results are significant (p < 0.05). Its improvements over DDFuzz are significant except for libxml2.

Benchmark	StorFuzz	LibAFL	DDFuzz
bloaty	7 556.5 (+19.5)	7 538.5 (+1.5)	7 538 (+1)
curl	11 569.5 (+15.5)	11 557 (+3)	11 557 (+3)
freetype2	18 054.5 (+11.5)	18 044.5 (+1.5)	18 045 (+2)
harfbuzz	21 870.5 (+24.5)	21 847.5 (+1.5)	21 854.5 (+8.5)
libxml2	16 314 (+3)	16 311.5 (+0.5)	16 313 (+2)
libxslt	12 158.5 (+23.5)	12 143.5 (+8.5)	12 139 (+4)
mbedtls	4 419 (+1)	4 418 (+0)	4 418 (+0)
openh264	9 841 (+2)	9 839 (+0)	9 839 (+0)
openssl	5 977 (+3)	5 977 (+3)	5 977 (+3)
openthread	4 651 (+70)	4 637.5 (+56.5)	4 583.5 (+2.5)
proj4	11 119 (+9)	11 112 (+2)	11 111 (+1)
re2	2 938 (+1)	2 938 (+1)	2 938 (+1)
sqlite3	21 592 (+47)	21 578.5 (+33.5)	21 582 (+37)



(a) bloaty

(b) openthread

Figure 4: Starting from the saturated corpus, diversifying data with StorFuzz (●) for 24h results in higher coverage than diversifying with DDFuzz (●) or restarting LibAFL (●). Results for all benchmarks are reported in Table 1.

We expect potential improvements for any fuzzer to be small in comparison to the total number of covered branches, as we start from an already saturated corpus. As such, much of the binaries' code that is reachable from the fuzzing harness should have been covered by the seed corpus already.

4.4.1 Results. StorFuzz performs equal to or better than LibAFL and DDFuzz for all 23 benchmarks. It achieves higher coverage improvements than LibAFL and DDFuzz in 11 cases, the same improvements in 2 cases (openssl and re2), and in 10 cases no fuzzer can achieve any new coverage. We report the median covered branches across all 10 trials for each benchmark in Table 1. Figure 4 shows code coverage over time for bloaty and openthread.

The results show that StorFuzz manages to reach code parts that have not been covered by state-of-the-art code coverage-guided fuzzers in a long time. Moreover, our increased diversity in the equivalence classes also serves as a stepping stone for subsequent conventional fuzzers, as we show in Section 4.6.

4.5 Assessing Scale of Diversity

We analyze the sensitivity of our data coverage metric by comparing corpus growth. A more sensitive metric introduces more diversity into the corpus, because it can detect smaller changes in behavior. Therefore, we compare the corpus size reached by StorFuzz after 24 hours to the corpus sizes of LibAFL and DDFuzz after 24 hours.

4.5.1 Results. StorFuzz generally produces larger corpora than LibAFL and DDFuzz, indicating that our novel data coverage is more sensitive than edge coverage and data-flow coverage.

After 24 hours of fuzzing to diversify the already saturated seed corpus, StorFuzz creates a corpus up to 8.6-times larger than the seed set it started from, showing that StorFuzz can diversify even saturated corpora (mbedtls). At the same time, the corpus size is reduced for other benchmarks, like libxml2 or libxslt. This indicates that the seed set contained test cases that duplicated both code and data coverage. The reduction further shows that our metric can effectively differentiate behavior. For LibAFL, the corpus is smaller than the seed set for all benchmarks, meaning the seeds contained test cases with the same code coverage. DDFuzz shrinks the corpus for most benchmarks, except freetype2, which indicates that data-flow coverage is largely not more sensitive than edge coverage. Compared to StorFuzz, DDFuzz is less sensitive for all benchmarks except re2. This aligns with our previous results (see Section 4.4.1), where DDFuzz performs comparable to StorFuzz on re2. When we account for differences in throughput, DDFuzz instrumentation also adds more test cases to the corpus for sqlite3, but cannot materialize this due to significantly slower execution speed. All statistics are available as part of our artifact [35].

Overall, our data coverage metric introduced with StorFuzz is more sensitive than the coverage metrics of LibAFL and DDFuzz. Consequently, using StorFuzz results in a higher seed set diversity.

4.6 Transferring the Diversity: LibAFL

StorFuzz achieves higher seed diversity than LibAFL and DDFuzz, but it is not yet clear that this increased diversity enables conventional code coverage-guided fuzzers to escape the plateau themselves. Next, we investigate whether diversifying the saturated corpus using StorFuzz for 24 hours and then letting a conventional fuzzer run for another 24 hours results in higher code coverage than continuing to fuzz with a conventional fuzzer for the same amount of time or restarting the conventional fuzzer.

As baseline, we run LibAFL for an additional 4 days (96 hours) on the saturated corpus, effectively creating a 120 + 96 hours-long run. We then compare the resulting code coverage to two separate 96 hours-long campaigns that start with the saturated corpus as seed sets: (1) a campaign switching twice between 24 hours of diversification with StorFuzz and 24 hours of code coverage-guided fuzzing with LibAFL, to determine the improvements added to LibAFL by StorFuzz, and (2) a campaign that restarts LibAFL at the same interval, to account for the positive effect of restarting [27].

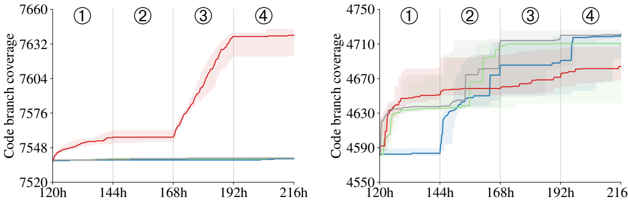
4.6.1 Results. Our results show that both fuzzer restarts and our corpus diversification improve code coverage, with StorFuzz achieving equal or higher improvements in all but two cases.

Table 2 shows the coverage achieved by the individual fuzzer configurations. We switch between the fuzzers at the indicated times, that is, StorFuzz/LibAFL runs StorFuzz 120h–144h and 168h–192h, and LibAFL 144h–168h and 192h–216h, while LibAFL/LibAFL only runs LibAFL, but restarts it at 120h, 144h, 168h, and 192h.

Table 2: Branches covered at different times. For each benchmark, the first row is restarting LibAFL, and the second row is StorFuzz/LibAFL. We report the median over 10 trials and changes to the previous point, omitting unchanged values. Green (□) marks the best configurations at each point.

	Code Coverage after			
120 h	① 144 h	② 168 h	③ 192 h	④ 216 h
Benchmark	LibAFL StorFuzz	LibAFL LibAFL	LibAFL StorFuzz	LibAFL LibAFL
bloaty	7 538.5 (+1.5)	7 539 (+0.5)	7 539.5 (+0.5)	
	7 556.5 (+19.5)		7 638 (+81.5)	7 639 (+1)
curl	11 557 (+3)	11 558 (+1)	11 558.5 (+0.5)	
	11 569.5 (+15.5)	11 571 (+1.5)	11 574.5 (+3.5)	11 575 (+0.5)
freetype2	18 044.5 (+1.5)	18 045 (+0.5)	18 046 (+1)	18 046.5 (+0.5)
	18 054.5 (+11.5)	18 055 (+0.5)	18 060 (+5)	18 060.5 (+0.5)
harfbuzz	21 847.5 (+1.5)	21 852 (+4.5)	21 867 (+15)	21 877 (+10)
	21 870.5 (+24.5)	21 873.5 (+3)	21 878 (+4.5)	21 889 (+11)
libxml2	16 311.5 (+0.5)	16 314 (+2.5)	16 315 (+1)	16 315.5 (+0.5)
	16 314 (+3)	16 314.5 (+0.5)	16 319 (+4.5)	16 321 (+2)
libxslt	12 143.5 (+8.5)	12 144.5 (+1)		12 145 (+0.5)
	12 158.5 (+23.5)	12 160 (+1.5)	12 162 (+2)	12 163 (+1)
mbedtls	4 418 (+0)			
	4 419 (+1)			
openh264	9 839 (+0)			
	9 841 (+2)			
openssl	5 977 (+3)			
	5 977 (+3)	5 977.5 (+0.5)	5 979 (+1.5)	
openthread	4 637.5 (+56.5)	4 712 (+74.5)	4 720 (+8)	4 721.5 (+1.5)
	4 651 (+70)	4 658.5 (+7.5)	4 671.5 (+13)	4 684 (+12.5)
proj4	11 112 (+2)	11 113 (+1)	†	11 114† (+1)
	11 119 (+9)	11 120 (+1)	11 125.5 (+5.5)	11 127.5 (+2)
re2	2 938 (+1)			‡
	2 938 (+1)			‡
sqlite3	21 578.5 (+33.5)	21 601.5 (+23)	21 614 (+12.5)	21 622.5 (+8.5)
	21 592 (+47)	21 609 (+17)	21 623.5 (+14.5)	21 630.5 (+7)

† Continuous LibAFL outperforms LibAFL/LibAFL by 2 branches after 192h and 2.5 branches after 216h. StorFuzz/LibAFL still performs best. ‡ DDFuzz/LibAFL is able to reach 2939 branches after 216h, outperforming the others by 1 branch.



(a) bloaty (b) openthread

Figure 5: Diversification with StorFuzz (●) outperforms restarting LibAFL (●) for most benchmarks. For not fully saturated benchmarks, like openthread, restarting LibAFL initially performs worse, but is able to surpass all other fuzzers later on. Diversification phases with StorFuzz (①/③) are followed by conventional fuzzing (LibAFL) (②/④). Vertical lines indicate switches/restarts. Full results including DDFuzz (●) and continuous LibAFL (●) are in our artifact.

Table 3: Branches covered at different points during the fuzzing campaigns. For each benchmark, the first row is StorFuzz/LibAFL, and the second row is WingFuzz/libFuzzer. We report the median over 10 trials and relative changes to the previous point, omitting unchanged values. Green (🟩) marks the best performing configurations at each point.

	Code Coverage after			
120 h	① 144 h	② 168 h	③ 192 h	④ 216 h
Benchmark	StorFuzz WingFuzz	LibAFL libFuzzer	StorFuzz WingFuzz	LibAFL libFuzzer
bloaty	7 565.5 (+15.5)	7 566.5 (+1)	7 596.5 (+30)	
	🟩 7 628.5 (+78.5)	🟩 7 629 (+0.5)	🟩 7 651 (+22)	🟩 7 651.5 (+0.5)
curl	11 633.5 (+10.5)	11 634.5 (+1)		11 635 (+0.5)
	🟩 11 636 (+13)	🟩 11 637.5 (+1.5)	🟩	🟩 11 639 (+1.5)
freetype2	18 126 (+19)	18 129 (+3)	18 129.5 (+0.5)	
	🟩 18 134 (+27)	🟩 18 135 (+1)	🟩 18 136 (+1)	🟩
harfbuzz	🟩 21 908 (+29)	🟩 21 914.5 (+6.5)	🟩 21 922 (+7.5)	🟩 21 927.5 (+5.5)
	21 881 (+2)	21 904 (+23)	21 910.5 (+6.5)	21 913.5 (+3)
lcms	🟩 2 476 (+0)	🟩 2 480 (+4)	🟩	🟩
	🟩 2 476 (+0)	2 476.5 (+0.5)	2 477 (+0.5)	
libpcap	🟩 4 469 (+0)			
	🟩 4 469 (+0)	🟩 4 469.5 (+0.5)	🟩 4 470 (+0.5)	🟩
libxml2	16 394 (+0)	16 395 (+1)	16 396 (+1)	16 397.5 (+1.5)
	🟩 16 396.5 (+2.5)	🟩	🟩 16 400 (+3.5)	🟩 16 401 (+1)
libxslt	🟩 12 231 (+14)	🟩 12 236.5 (+5.5)	🟩 12 237 (+0.5)	🟩 12 239 (+2)
	12 222 (+5)	12 226.5 (+4.5)	12 231 (+4.5)	12 234.5 (+3.5)
openssl	5 985 (+3)			
	🟩 5 985.5 (+3.5)	🟩 5 986 (+0.5)	🟩 5 986.5 (+0.5)	🟩 5 987.5 (+1)
openthread	🟩 4 791 (+13)	🟩 4 795 (+4)	🟩 4 799 (+4)	🟩 4 800.5 (+1.5)
	4 780 (+2)	4 781 (+1)		4 781.5 (+0.5)
proj4	11 356 (+5)	11 362 (+6)	11 369.5 (+7.5)	11 370 (+0.5)
	🟩 11 371.5 (+20.5)	🟩 11 377 (+5.5)	🟩 11 391.5 (+14.5)	🟩 11 397 (+5.5)
re2	2 963 (+0)			
	🟩 2 965 (+2)	🟩	🟩	🟩 2 966 (+1)
sqlite3	🟩 21 682.5 (+97.5)	🟩 21 694.5 (+12)	🟩 21 707 (+12.5)	🟩 21 715 (+8)
	21 665.5 (+80.5)	21 678.5 (+13)	21 695 (+16.5)	21 703 (+8)
vorbis	1 381 (+0)			
	🟩 1 384 (+3)	🟩	🟩	🟩

StorFuzz/LibAFL is statistically significantly better in all cases where any fuzzer gains at least one branch, except for openthread. Comparing the LibAFL/LibAFL restart configuration to switching between StorFuzz and LibAFL, diversifying with StorFuzz leads to a larger improvement in 11 of 23 benchmarks, all of which are statistically significant (Mann-Whitney-U, p<0.05). Restarting only performs better for openthread, for which coverage increased up until 120 hours in the saturation experiment (see Section 4.3.1). This may explain how LibAFL could progress with code coverage and mutation operations alone.

Looking at Fig. 5a, we see that StorFuzz is beneficial beyond the initial diversification. For example, for bloaty (Fig. 5a) there are clearly visible gains in day ① and ③, when StorFuzz is running, that cannot be observed for LibAFL (②/④).

StorFuzz outperformed DDFuzz in 11/13 benchmarks for which any fuzzer improved upon the initial coverage (see Section 4.4). This trend continues when replacing StorFuzz with DDFuzz in this transfer experiment. Similarly, continuous LibAFL performs worse than LibAFL with restarts, replicating prior work [27]. Thus, for clarity, we omit the DDFuzz and continuous LibAFL results here and refer to our artifact for more detailed results [35].

Table 4: Diversification with StorFuzz covers a different set of branches than diversifying with WingFuzz. We present data for branches covered in at least 1, 3, and 5 trials to avoid bias from a single successful trial. We omit zeros.

Covered in *n* trials	**Only by StorFuzz**			**Only by WingFuzz**		
	1+	3+	5+	1+	3+	5+
bloaty	24	6	2	58	61	66
curl	14	6	4	18	8	8
freetype2	8	4	1	21	13	9
harfbuzz	51	30	37	15	8	3
lcms	4	4	4	1	1	1
libpcap				2	1	
libxml2	13	5	3	26	6	5
libxslt	28	22	11	23	14	7
openh264				3		
openssl				17	2	1
openthread	48	25	25	11	2	1
proj4	4	12	11	118	50	25
re2				7	4	4
sqlite3	88	31	30	128	31	28
systemd				1	1	
vorbis				3	3	3

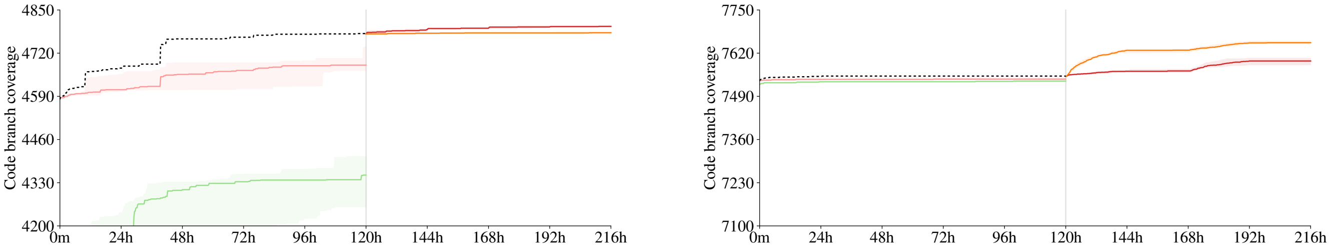
4.7 Transferring the Diversity: WingFuzz

Following, we compare StorFuzz against the state-of-the-art data coverage-guided fuzzer WingFuzz [33], which focuses on the use of constants and static values. WingFuzz is, however, based on libFuzzer, which would make a comparison against LibAFL-based fuzzers unfair when the corpus is predominantly saturated with LibAFL. To investigate the coverage metric independent of other fuzzer design choices, like mutation operations, we first need a corpus that is saturated both by LibAFL and libFuzzer.

To obtain the saturated corpus, we extend the OSS-Fuzz corpus with five 120-hours libFuzzer trials (like Section 4.3) and merge the results with our LibAFL-saturated corpora. Starting from our merged seed corpus, we conduct another transfer experiment (see also Section 4.6) by diversifying with StorFuzz or WingFuzz for 24 hours, followed by 24 hours of fuzzing with their respective base fuzzer. We switch between diversification and conventional code coverage-guided fuzzing twice.

Figure 6 shows how the coverage develops in the saturation and fuzzing phases. Combining the corpora from libFuzzer and LibAFL results in higher coverage than either of them achieves on their own, highlighting that there are indeed differences in the base fuzzers. This confirms that saturating with both is necessary to fairly assess the capabilities of fuzzers derived from them.

4.7.1 Results. StorFuzz and WingFuzz both continue to make progress on the corpora already saturated by LibAFL and libFuzzer. Table 3 shows that neither fuzzer has a general advantage over the other. StorFuzz performs better in 5 of 14 benchmarks that see any improvement for either of the two fuzzers, while WingFuzz covers more branches in the median case for 9 out of 14. For a more detailed analysis of this behavior, we investigate the branches that are uniquely covered by either of the two fuzzers. The data shows that StorFuzz and WingFuzz systematically explore different code regions, even within benchmarks where one fuzzer outperforms the other, like StorFuzz does for openthread and WingFuzz for bloaty.



(a) openthread

(b) bloaty

Figure 6: Both diversifying with StorFuzz (●) and WingFuzz (●) is beneficial. The individual efficacy is target dependent, both cover a distinct set of branches (see Table 4). In the time up until the vertical line at 5 days, LibAFL (●) and libFuzzer (●) are running separately, creating the saturated corpora. The combined coverage of both libFuzzer and LibAFL is shown dashed. After the vertical line, StorFuzz/LibAFL (●) and WingFuzz/libFuzzer (●) are run for 96 hours, starting with the merged corpora.

This pattern remains consistent when we account for the inherent randomness of fuzzing by only considering branches covered in at least three or five trials (see Table 4). In fact, this clearly shows that there is a systemic difference in the coverage approaches employed by the two fuzzers, making them complementary and indicating the need for both in comprehensive long-term fuzzing.

4.8 Performance Impact

Gathering and recording our novel data coverage requires additional instructions and memory. For each store instrumentation point, we add at most 12 instructions, when LLVM cannot optimize further. For the FuzzBench benchmarks, we have 2,485 instrumented locations for the median case, which results in less than 30,000 added instructions. StorFuzz's memory overhead is fixed at 256 kB for our map size, as StorFuzz requires 128 kB for the current map plus an additional 128 kB for the historic data map.

We conducted a performance ablation study comparing regular StorFuzz to a variant that does not include data coverage on ten 1-hour trials for all 23 FuzzBench targets. The data coverage of StorFuzz introduces about 46% overhead for executions per second. This is similar to the overhead reported for cmplog and input2state correspondence [3], which are included in LibAFL's state-of-the-art configuration for code coverage. We believe that the significant coverage improvements over other fuzzers post-plateau, which we have shown in the previous sections, justify this cost.

4.9 Ablation Study

To verify the design decisions limiting the number of stores per basic block, ignoring memory-to-memory copies and reducing each value down to 8 bits, we conducted an ablation study. Our experiment consisted of 10 one-day trials for each of the following alternative configurations, which we compared to the baseline StorFuzz that we propose in this paper: (1) reduction of each value to 4, 12, or 16 bits, (2) limiting the number of stores to 1, 5, 20 per BB, or not limiting them at all, and (3) including memory-to-memory copies. Each configuration differs from our baseline only by changing the respective threshold or omitting the relevant check. We list the alternative reduction functions in the artifact [35].

The results of our ablation study experiment confirmed our individual design choices, as the presented configuration performed best when considering the average rank across all 23 FuzzBench benchmarks. For some individual benchmarks, alternative configurations performed better (e.g., reducing to 12 bits for harfbuzz or openssl; or to 4 bits for libxml2 or proj4). However, as StorFuzz is designed to be a generic solution, we did not tune parameters for individual targets but instead use the overall best configuration to allow a fair evaluation on FuzzBench. Practitioners fuzzing specific targets should determine whether our defaults are optimal for them.

Our ablation study also highlights another trade-off: Increasing the number of bits of the value reduction leads to a drastically decreased execution speed. This is expected because we now need a significantly larger map to account for all possible values, which has to be evaluated by the fuzzer for each test case. Across all benchmarks, increasing the reduced value to 12 bits results in a slowdown of more than 40% (>95% for 16 bits) and decreasing the value's width to 4 bits increases the execution speed up to 5x, though this does not lead to additional coverage. When comparing the corpus growth of the alternative reduction functions to the one described in Listing 1, we confirm the intuition that functions reducing to higher bit-widths are more sensitive.

For the other configuration variants, the effects do not generalize across all benchmarks, indicating that the optimal choices might depend on the specific target.

4.10 Bug Finding Case Study

To showcase StorFuzz's bug-finding capabilities on a larger set of programs that are fuzzed continuously, we targeted open-source C and C++ projects in OSS-Fuzz [1]. We ranked them by their GitHub forks and stars as a measure of popularity and selected the top 100 most-forked and top 100 most-starred ones, resulting in 122 projects. We enabled ASAN and attempted to build all of them with StorFuzz and LibAFL. Of these 122 projects, 83 successfully compiled fuzzing harnesses without manual intervention.

Similar to our coverage evaluation setup, we started from an already saturated corpus (here, the latest public OSS-Fuzz corpus). For each harness, we first run StorFuzz for 24 hours on a single core to diversify the corpus, followed by 24 hours of fuzzing with LibAFL on a single core. We triaged crashes using AFLTriage [13], then manually assessed the AFLTriage reports. We list short descriptions of every bug discovered with StorFuzz in our artifact [35].

4.10.1 Statistics. We discovered 50 new, previously unknown bugs in 7 well-fuzzed OSS projects. These numbers do not include crashes found in another 17 projects that we can attribute to issues in their fuzzing harnesses. Among these new bugs, there were *previously unknown exploitable vulnerabilities in projects like VLC [31] and the Open Asset Importer Library (assimp) [29]*, as well as a bug in PHP [30]. Additionally, StorFuzz rediscovered at least 20 known, but unresolved issues, like CVE-2024-48426.

We reported the bugs to the developers through coordinated disclosure. In coordination with the maintainers, 21 new CVEs were assigned. All issues in release versions have been fixed.

4.10.2 Case Studies. Most projects in which we discovered bugs support a variety of (complex) file formats, often with numerous variants and extensions. VLC and assimp are two examples of such projects. Despite having been fuzzed in OSS-Fuzz for more than 4 years, StorFuzz could still find new bugs in multiple formats. In fact, one bug (vlc-7) has been present in the code since at least 2011, highlighting the insufficiency of current state-of-the-art code coverage-guided fuzzers.

We looked further into the areas that had previously been covered by the OSS-Fuzz corpus and newly covered by StorFuzz. Interestingly, for assimp, OSS-Fuzz had already partially fuzzed most of the buggy functions. However, StorFuzz covered state-dependent parts that were previously not covered (e.g., assimp-1).

In other cases, OSS-Fuzz had covered the buggy lines already but could not setup the states required to trigger the bugs. One of these bugs is vlc-6, an out-of-bounds write into an array on the stack. To trigger this bug, the fuzzer must construct a WAV audio file with two format chunks. The first chunk's audio channel configuration is used to create a specific channel mask (a value tracked by StorFuzz). This is turned into a channel table that maps the file's audio channels to VLC's internal representation. When the second chunk is parsed, the table is not updated. If the second chunk reports more channels than the first, uninitialized entries in the table will be used as indices and cause an out-of-bounds write. StorFuzz is particularly well-suited to discovering this type of bug because it can incrementally build on test cases with different channel masks and channel tables, while typical coverage-guided fuzzers discard most of those inputs.

5 Discussion

Our experiments show that StorFuzz successfully escapes the fuzzing plateau and achieves higher code coverage gains compared to state-of-the-art code coverage-guided fuzzers, like LibAFL, when starting from a saturated corpus. Our results further show that StorFuzz reaches more code than other fuzzers that employ more sensitive metrics, such as data-flow coverage. Even for cases where StorFuzz does not cover strictly more code than fuzzers using different data coverage techniques, it covers a distinct set of branches. This shows that our novel approach is complementary to existing techniques, which makes it a valuable contribution.

Based on our observations in our coverage and sensitivity experiments, combined with our bug case study, we can conclude that the way StorFuzz incorporates stored values is responsible for the observed coverage improvements. With its novel data feedback, StorFuzz retains multiple test cases per code coverage equivalence class that result in different states depending on the stored values. StorFuzz can effectively distinguish different states, which is further highlighted by our bug-finding experiment, where OSS-Fuzz fuzzers covered some of the buggy lines before but were not able to set up the necessary state for them to crash (e.g., vlc-6).

Not all classes will be diversified equally, because different paths through the program have different amounts of stores associated with them and different store instructions have varying ranges of distinct values they store. Fuzzers are more likely to choose from larger classes, putting higher emphasis on code paths with more varied store characteristics. Thus, StorFuzz implicitly favors test cases from equivalence classes with more opportunities to influence state and explores these classes more thoroughly than others.

However, this also means that the effectiveness of our approach may decrease when StorFuzz considers too many stores of values that do not contribute meaningfully to the state. Examples are the current time, random values, or parts of the input. In this case, StorFuzz may diversify some equivalence classes more than necessary, putting too much emphasis on the contained code paths. We did not encounter this to be a problem in our experiments, but this might be different for other targets, and future work should investigate how to further improve data coverage metrics.

In our experiments, we observed that diversifying the fuzzing corpus with additional, more verbose data coverage enables conventional coverage-guided fuzzers to more effectively overcome plateaus in fuzzing campaigns. Plotting coverage over time showed that the gain for some of the benchmarks happens over a short period of time, and thus, we can conclude that StorFuzz could also be used to improve the results of long-running fuzzing campaigns by only being a short intervention rather than letting it run for 24 hours. Furthermore, some targets reached their plateau earlier than others, providing an opportunity for better use of resources by switching to diversification with StorFuzz earlier. Future work should look into detecting the plateau sooner and investigate for how long StorFuzz needs to run for a sufficient amount of diversity.

Wang et al. [32] have postulated that the number of seeds may not only be a measure of sensitivity but too many seeds may be harmful to performance when their differences are too small. In extreme cases of our experiment, the corpus generated by StorFuzz was 8.6x larger than the saturated corpus. According to their hypothesis, this should result in degraded performance of fuzzers starting from it. However, our experiments show the opposite. In fact, even though many of the inputs share equivalence classes (have the same code coverage), fuzzers starting from the StorFuzz-diversified corpus reach higher coverage than those starting from the corpus created by fuzzing with LibAFL in 11/23 cases (in 11 more cases they show equal coverage).

Our results highlight the importance of data diversity in fuzzing and its benefits alongside traditional code coverage.

5.1 Limitations and Threats to Validity

Our StorFuzz prototype instruments stores in a generic fashion. We chose to reduce integers to 8 bits to improve fuzzing throughput and control the coverage granularity. Tailoring parameter choices to the target could further improve code coverage. We discussed the impact of differently-sized coverage equivalence classes, but further analysis of test case distribution could give additional insights into the effects that our data coverage metric has beyond edge coverage.

We performed 10 trials for all experiments following current best practices; additional trials could further increase the confidence in our results. While we report statistical significance measures, it is not clear if coverage distributions of fuzzing campaigns satisfy all assumptions of the Mann-Whitney U test. Diversification might also lead to different results depending on the initially plateauing fuzzer. We address this concern by (1) starting with the already diverse corpus of OSS-Fuzz, which includes libFuzzer, honggfuzz, and AFL++, and (2) including an additional evaluation against WingFuzz, which starts from a corpus further diversified by libFuzzer.

Evaluating bug-finding capabilities is known to be difficult, and only a limited set of bug-based benchmarks exists [5, 7, 12, 36]. As StorFuzz requires a saturated corpus, these benchmarks are unsuitable, as either no corpora exist, or fuzzing with a coverage-guided fuzzer for a short time already finds most injected bugs.

6 Related Work

Breaking out of Fuzzing Plateaus. Schiller et al. [27] propose fuzzer restarts to break out of fuzzing plateaus. We have seen that periodically restarting the fuzzer can indeed help escape local optima because it has a chance to retain a test case with a better placement in the code coverage equivalence class. However, StorFuzz outperforms it with its capability to keep multiple test cases per class and make incremental progress towards new classes (see Fig. 1). Lemieux et al. [18] discover that querying Large Language Models (LLMs) helps overcome coverage plateaus in search-based software testing of Python modules. Adapting input generation is an orthogonal approach to StorFuzz and LLM-based test case generation has also not yet been successfully demonstrated for programs expecting binary data.

Data Guided Fuzzing. Prior work has also explored using data to guide fuzzing and improve its effectiveness. GREYONE [11] and VUzzer [26] employ forms of taint-tracking to guide mutations based on variables' values. Others use the locations of memory accesses instead of the concrete values to produce additional coverage information during fuzzing [6, 32]. With WingFuzz [33], we have seen that approaches guided by constant data that is used for comparisons can also diversify the corpora, but we have shown that StorFuzz covers different areas. DatAFLow and DDFuzz [14, 22] take a different approach by considering def-use chains in their guidance metric, however StorFuzz proved more effective.

Explicit State Recovery. Some approaches attempt to explicitly recover state variables or full state machines. In contrast to StorFuzz, this typically requires much more expensive analysis and additional assumptions on how the target program represents states. For example, SandPuppy [25] and StateFuzz [40] are specifically designed to identify persistent state variables,or more precisely, in the case of StateFuzz, data-dependencies to global variables with def-use relationships across program actions. Ba et al. [4] extract state machines from network protocol implementations by examining uses of enumerated types in the target's code base. DSFuzz [21] constructs a full state graph of the target by analyzing control and data dependencies between values and then attempts to guide a fuzzer along the recovered state transitions. Ferry [41] identifies state by symbolic taint tracking and then builds a state graph by exploring possible state transitions using symbolic execution. With its novel data coverage, StorFuzz is a more generic approach that makes fewer assumptions and complements code coverage-guided fuzzers when they are plateaued.

7 Conclusion

In this paper, we introduce StorFuzz, a novel approach to overcome fuzzing plateaus using our new data coverage metric focusing on store operations. StorFuzz successfully escapes coverage plateaus in stagnating fuzzing campaigns, a challenging problem of fuzzing in the real world, by diversifying the saturated corpus. It enables state-of-the-art code coverage-guided fuzzers to achieve higher code coverage by leveraging data coverage and providing insight into memory stores. This allows StorFuzz to capture changes in program behavior and state that do not (yet) manifest in control flow changes and, thus, cannot be observed by existing fuzzers. Using these novel insights, StorFuzz improves navigation of the input space because it now considers test cases, which have not covered new code yet, but can be incrementally transformed into new test cases that reach new code areas using conventional mutations.

In our evaluation on all 23 FuzzBench benchmarks, we have shown StorFuzz's effectiveness starting from a saturated corpus. StorFuzz efficiently generates new coverage for plateauing campaigns, and StorFuzz performs equal or better than the state-of-the-art fuzzer LibAFL, achieving the same or more new code coverage for 22 out of 23 benchmarks, strictly outperforming LibAFL for 11. Even compared to more advanced configurations, such as fuzzer restarts, or diversification with other fine-grained fuzzers that consider data-dependencies or comparison operations, it discovers more code or covers a distinct set of branches that existing fuzzers miss. Moreover, StorFuzz's code coverage and corpus diversity improvements transfer to subsequent fuzzing campaigns of conventional fuzzers without requiring any changes to them.

Leveraging StorFuzz, we discovered new bugs in real-world software, finding 50 unique, previously unknown bugs in 7 widely-fuzzed open-source projects, with some of the bugs having been undetected for 14 years, and 21 CVEs have been assigned so far.

We provide StorFuzz and our artifacts as open source [34, 35].

Acknowledgments

We thank our anonymous reviewers for their constructive and insightful comments. We also thank Marcel Böhme for his valuable feedback on an earlier version of this paper.

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

References

[1] A. Arya, O. Chang, J. Metzman, K. Serebryany, and D. Liu. *OSS-Fuzz*. Version e17999e. Dec. 19, 2024. URL: https://github.com/google/oss-fuzz (visited on 12/20/2024).

[2] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz. "Ijon: Exploring Deep State Spaces via Fuzzing." In: *Proceedings of the 41st IEEE Symposium on Security & Privacy (S&P)*. May 2020. DOI: 10.1109/SP40000.2020.00117.

[3] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz. "REDQUEEN: Fuzzing with Input-to-State Correspondence." In: *Proceedings of the 26th Network and Distributed System Security Symposium (NDSS)*. Feb. 2019. DOI: 10.14722/ndss.2019.23371.

[4] J. Ba, M. Böhme, Z. Mirzamomen, and A. Roychoudhury. "Stateful Greybox Fuzzing." In: *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*. Aug. 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/ba.

[5] B. Caswell. *Cyber Grand Challenge Corpus*. Lunge Technology. Apr. 1, 2017. URL: http://www.lungetech.com/cgc-corpus/ (visited on 01/08/2025).

[6] N. Coppik, O. Schwahn, and N. Suri. "MemFuzz: Using Memory Accesses to Guide Fuzzing." In: *Proceedings of the 12th IEEE/ACM International Conference on Software Engineering (ICSE)*. Apr. 2019. DOI: 10.1109/ICST.2019.00015.

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

[8] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. "AFL++: Combining Incremental Steps of Fuzzing Research." In: *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT)*. Aug. 11, 2022. URL: https://www.usenix.org/conference/woot20/presentation/fioraldi.

[9] A. Fioraldi, D. Maier, D. Zhang, and D. Balzarotti. "LibAFL: A Framework to Build Modular and Reusable Fuzzers." In: *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Nov. 2022. DOI: 10.1145/3548606.3560602.

[10] *Fuzzing Introspection of OSS-Fuzz projects*. URL: https://introspector.oss-fuzz.com/ (visited on 03/10/2025).

[11] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen. "GREYONE: Data Flow Sensitive Fuzzing." In: *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*. Aug. 2020. DOI: 10.5555/3489212.3489357.

[12] A. Hazimeh, A. Herrera, and M. Payer. "Magma: A Ground-Truth Fuzzing Benchmark." In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems (SIGMETRICS)* 4 (3 Dec. 2020). DOI: 10.1145/3428334.

[13] G. Hernandez. *AFLTriage*. Version e86d323. Qualcomm Innovation Center, Inc., Jan. 14, 2022. URL: https://github.com/quic/AFLTriage (visited on 12/23/2024).

[14] A. Herrera, M. Payer, and A. L. Hosking. "DatAFLow: Toward a Data-Flow-Guided Fuzzer." In: *ACM Transactions on Software Engineering and Methodology* 32.5 (July 2023). DOI: 10.1145/3587156.

[15] *Honggfuzz*. URL: https://github.com/google/honggfuzz (visited on 07/06/2024).

[16] T. Klooster, F. Turkmen, G. Broenink, R. T. Hove, and M. Böhme. "Continuous Fuzzing: A Study of the Effectiveness and Scalability of Fuzzing in CI/CD Pipelines." In: *Proceedings of the 2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*. May 2023. DOI: 10.1109/SBFT59156.2023.00015.

[17] lafintel. *Circumventing Fuzzing Roadblocks with Compiler Transformations*. Aug. 2016. URL: https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/ (visited on 07/27/2023).

[18] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen. "CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-Trained Large Language Models." In: *Proceedings of the 2023 45th IEEE/ACM International Conference on Software Engineering (ICSE)*. May 2023. DOI: 10.1109/ICSE48619.2023.00085.

[19] *libFuzzer – a library for coverage-guided fuzz testing*. URL: https://github.com/llvm/llvm-project/blob/main/llvm/docs/LibFuzzer.rst (visited on 07/11/2024).

[20] S. Lipp, D. Elsner, S. Kacianka, A. Pretschner, M. Böhme, and S. Banescu. "Green Fuzzing: A Saturation-Based Stopping Criterion using Vulnerability Prediction." In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. July 2023. DOI: 10.1145/3597926.3598043.

[21] Y. Liu and W. Meng. "DSFuzz: Detecting Deep State Bugs with Dependent State Exploration." In: *Proceedings of the 30th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Nov. 2023. DOI: 10.1145/3576915.3616594.

[22] A. Mantovani, A. Fioraldi, and D. Balzarotti. "Fuzzing with Data Dependency Information." In: *Proceedings of the 7th IEEE European Symposium on Security & Privacy (S&P)*. June 2022. DOI: 10.1109/eurosp53844.2022.00026.

[23] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya. "FuzzBench: An Open Fuzzer Benchmarking Platform and Service." In: *Proceedings of the 29th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. Aug. 2021. DOI: 10.1145/3468264.3473932.

[24] R. Natella. "StateAFL: Greybox Fuzzing for Stateful Network Servers." In: *Empirical Software Engineering* 27.7 (2022). DOI: 10.1007/S10664-022-10233-3.

[25] V. Paliath, E. Trickel, T. Bao, R. Wang, A. Doupe, and Y. Shoshitaishvili. "SandPuppy: Deep-State Fuzzing Guided by Automatic Detection of State-Representative Variables." In: *Proceedings of the 21st Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. July 2024. DOI: 10.1007/978-3-031-64171-8_12.

[26] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. "VUzzer: Application-aware Evolutionary Fuzzing." In: *Proceedings of the 24th Network and Distributed System Security Symposium (NDSS)*. Feb. 2017. DOI: 10.14722/ndss.2017.23404.

[27] N. Schiller, X. Xu, L. Bernhard, N. Bars, M. Schloegel, and T. Holz. "Novelty Not Found: Adaptive Fuzzer Restarts to Improve Input Space Coverage (Registered Report)." In: *Proceedings of the 2nd International Fuzzing Workshop (FUZZING)*. July 17, 2023. DOI: 10.1145/3605157.3605171.

[28] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz. "SoK: Prudent Evaluation Practices for Fuzzing." In: *Proceedings of the 45th IEEE Symposium on Security & Privacy (S&P)*. May 2024. DOI: 10.1109/SP54263.2024.00137.

[29] *The Open Asset Importer Library*. URL: https://assimp.org (visited on 01/08/2025).

[30] The PHP Group. *PHP*. URL: https://www.php.net/ (visited on 03/14/2025).

[31] *VLC media player*. URL: https://www.videolan.org/vlc/ (visited on 01/08/2025).

[32] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song. "Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing." In: *Proceedings of the 22nd International Symposium on Recent Advances in Intrusion Detection (RAID)*. Sept. 2019. URL: https://www.usenix.org/conference/raid2019/presentation/wang.

[33] M. Wang, J. Liang, C. Zhou, Z. Wu, J. Fu, Z. Su, Q. Liao, B. Gu, B. Wu, and Y. Jiang. "Data Coverage for Guided Fuzzing." In: *Proceedings of*

the 33rd USENIX Security Symposium (USENIX Security). Aug. 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/wang-mingzhe.

[34] L. Weiß, T. Holl, and K. Borgolte. *StorFuzz: Using Data Diversity to Overcome Fuzzing Plateaus — Fuzzing Corpora*. Mar. 2025. DOI: 10.5281/zenodo.14979693.

[35] L. Weiß, T. Holl, and K. Borgolte. *StorFuzz - Source Code and Full Data*. URL: https://github.com/rub-softsec/StorFuzz.

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

[37] D. Zhang (tokatoka). *DDFuzz - PR 2056*. Apr. 16, 2024. URL: https://github.com/AFLplusplus/LibAFL/pull/2056 (visited on 01/06/2025).

[38] D. Zhang (tokatoka). *Sancov based ngram & ctx implementation - PR 1864*. Feb. 16, 2024. URL: https://github.com/AFLplusplus/LibAFL/pull/1864 (visited on 03/14/2025).

[39] D. Zhang (tokatoka). *LibAFL commit bb579e6*. URL: https://github.com/AFLplusplus/LibAFL/commit/bb579e624e907b6488f019a6f0bb0634aa0f81da (visited on 09/04/2024).

[40] B. Zhao, Z. Li, S. Qin, Z. Ma, M. Yuan, W. Zhu, Z. Tian, and C. Zhang. "StateFuzz: System Call-Based State-Aware Linux Driver Fuzzing." In: *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*. Aug. 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/zhao-bodong.

[41] S. Zhou, Z. Yang, D. Qiao, P. Liu, Z. Wang, and C. Wu. "Ferry: State-Aware Symbolic Exeuction for Explorig State-Dependent Program Paths." In: *Proceedings of the 31st USENIX Security Symposium (USENIX Security)*. Aug. 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/zhou-shunfan.