Novelty Not Found: Exploring Input Shadowing in Fuzzing through Adaptive Fuzzer Restarts

NICO SCHILLER, CISPA, Germany
XINYI XU, CISPA, Germany
LUKAS BERNHARD, CISPA, Germany
NILS BARS, CISPA, Germany
MORITZ SCHLOEGEL, CISPA, Germany
THORSTEN HOLZ, CISPA, Germany

Greybox fuzzing enhances software security through unprecedented effectiveness in automated fault detection. Its success lies in the coverage feedback extracted from the system under test, guiding the fuzzer to explore different program parts. The most prominent way to use this feedback is *novelty search*, where the fuzzer keeps only new inputs exercising a new program edge. However, this approach—by design—ignores *input shadowing*, in which interesting inputs are discarded if they do not contribute to new coverage. This limits the accepted input space and may overlook bugs that shadowed inputs could trigger with mutations.

In this work, we present a comprehensive analysis of input shadowing and demonstrate that multiple fuzzing runs of the same target exhibit a different basic block hit frequency distribution despite overlapping code coverage. We propose *fuzzer restarts* to effectively redistribute basic block hit frequencies and show that this increases the overall achieved coverage on 15 evaluated targets on average by 9.5% and up to 25.0%. Furthermore, restarts help to find more bugs and trigger them more reliably. Overall, our results highlight the importance of considering input shadowing in the fuzzers' design and the potential benefits of a restart-based strategy to enhance the performance of complex fuzzing methods.

CCS Concepts: • **Security and privacy → Systems security**; **Software and application security**.

Additional Key Words and Phrases: fuzzing, fuzz testing

ACM Reference Format:
Nico Schiller, Xinyi Xu, Lukas Bernhard, Nils Bars, Moritz Schloegel, and Thorsten Holz. 2024. Novelty Not Found: Exploring Input Shadowing in Fuzzing through Adaptive Fuzzer Restarts. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (June 2024), 32 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Fuzzing is a highly effective technique for uncovering software faults in programs, and in recent years, it has gained rapid popularity in the software security testing community. Starting with feeding random inputs to Linux command line utilities and observing their behavior [44], fuzzing has evolved into a highly specialized, well-developed technique

Authors' addresses: Nico Schiller, CISPA, Germany, nico.schiller@cispa.de; Xinyi Xu, CISPA, Germany, xinyi.xu@cispa.de; Lukas Bernhard, CISPA, Germany, lukas.bernhard@cispa.de; Nils Bars, CISPA, Germany, nils.bars@cispa.de; Moritz Schloegel, CISPA, Germany, moritz.schloegel@cispa.de; Thorsten Holz, CISPA, Germany, holz@cispa.de.

that has undergone significant improvements in the past few years. Especially feedback-driven greybox fuzzing, as popularized by AFL [70], is one of the most significant advancements in fuzzing. This technique changes the approach from randomly generating inputs and observing the program to monitoring which parts of the program each input covered. This coverage guidance allows the fuzzer to effectively and efficiently select, mutate, and execute inputs that trigger new program behavior, maximizing the parts of the program that are tested.

To uncover as many bugs—and thus security vulnerabilities—as possible, a fuzzer attempts to test as much of the program's input space as it can. For this purpose, the fuzzer modifies a set of inputs using its mutators. Effectively, the fuzzer's mutators and the corpus shape the input space of the target that the fuzzer can cover. As fuzzers fundamentally rely on novelty search, they only accept inputs to the corpus if they exercise new code coverage within the target program. All inputs that exercise the same coverage are discarded, allowing the fuzzer to optimize its search to discover new program regions that have not been executed yet.

Unfortunately, this limits the input space that can be explored. Assume the fuzzer finds an input that hits new coverage in the program. The fuzzer considers this input to be novel and adds it to the corpus for further mutations. Now, the fuzzer finds another, different input, yet it exercises the same coverage; this second input will be discarded because it is not novel. We say the first input *shadows* the second one. However, the second input could be an essential precursor to reach an interesting program location later in the fuzzing campaign, even if no new coverage is explored. This is due to the fuzzer's inner workings: It derives the next input to test by selecting one input from the corpus and mutating it. As a result, the reachable input space is bounded by (i) the mutations a fuzzer can apply and (ii) the inputs in the corpus. This implies that parts of the input space remain unreachable (with non-negligible probability) to the fuzzer, as the available mutations applied to the corpus yield only a subset of all potential inputs. The existence of other, *shadowed* inputs in the corpus would increase the theoretically reachable input space. From this point of view, shadowed inputs should be included in the corpus to allow the fuzzer to explore a larger part of the input space. However, this is infeasible in terms of performance and collides with the principles of novelty search, calling for new techniques.

Previous work has explored two directions to address this problem. First, approaches such as CollAFL [22], datAFLow [29], or InvsCov [18] implicitly attempt to *circumvent* this problem by using more fine-grained or different feedback mechanisms. They increase the likelihood of considering previously shadowed inputs novel, according to their metric. On the other hand, increasing the number of inputs that are considered novel automatically causes the queue to grow, leading to the problem of wasting fuzzing cycles on irrelevant inputs. Even though these techniques decrease the number of shadowed inputs, they do not address the underlying problem. The second approach is techniques that bridge distant gaps in the input space, as depicted in Figure 1. This can either be achieved by dividing the gap into several smaller ones (e. g., via `cmplog` [3, 19]) or by improved mutations. Intuitively, we can use target-specific mutations to bridge distant gaps and, thus, increase the reachable input space [7, 37, 40]. While this may decrease the impact of shadowed inputs, it also does not address the underlying problem.

In this work, we analyze the impact of shadowed inputs on the explored input space and investigate a strategy to mitigate its effects: *fuzzer restarts*. By adaptively restarting the fuzzer, we reset its internal state and, thus, enable it to consider inputs as novel that were shadowed before. We hypothesize that fuzzers may benefit from such restarts in terms of coverage and bugs found, as this would undo previous decisions that may be blocking further fuzzer progress. While this appears counterintuitive, as information accumulated by costly executions is discarded, our results indicate that this technique is indeed beneficial. It diversifies the tested basic blocks and helps the fuzzer to find new coverage and trigger more bugs.

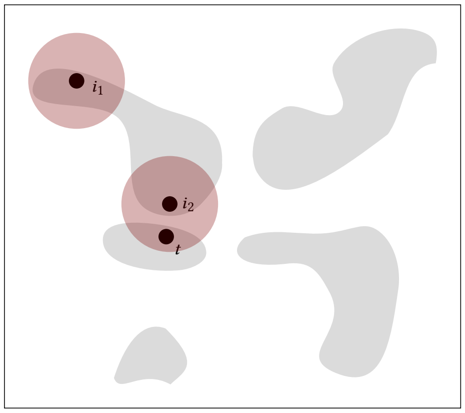


Fig. 1. Input space of a target program. Grey areas lead to the same coverage, red circles represent the input space we can reach with our mutations given the respective input. As illustrated, i_1 and i_2 exercise the same coverage, however, our mutations can derive input t only from i_2.

Contributions. In summary, we make the following three key contributions in this article:

- We shed light on the fact that fuzzing runs with the same initial configuration focus their attention on different program parts and frequently exhibit *input shadowing*, even though they achieve similar code coverage.
- To quantify this phenomenon, we propose *basic block frequency* as a new metric to measure the distribution of the fuzzer's attention across different program parts.
- Based on these insights, we present several scheduling strategies that reset the fuzzing state via fuzzer restarts, thereby mitigating input shadowing and achieving high block frequency as well as code coverage.

Research Artifact Availability. We publish our code and research artifacts at https://github.com/CISPA-SysSec/fuzzing-restarts to foster research in this area.

2 MOTIVATION

We start by discussing the concepts of *novelty search* and *input shadowing* and their impact on fuzzing progress.

2.1 Novelty Search in Fuzzing

A *novelty search algorithm* is an exploration algorithm driven by the observation of the novelty of a specific behavior [16]. In the field of fuzzing, novelty search is primarily used to determine if a given input is interesting for further mutations. For example, AFL uses a novelty search algorithm to decide whether an input should be kept by observing its yielded coverage during execution. The fast novelty search algorithm employed by AFL is one of its key features contributing to the success of its performance. Unfortunately, novelty search also comes with drawbacks, such as *input shadowing*.

2.2 Input Shadowing

Input shadowing describes the phenomenon that a coverage-guided fuzzer does not consider an input as novel because it does not contribute unique coverage. In practice, this means the fuzzer has found *another* input producing the *same* coverage first. On a technical level, most fuzzers track coverage progress using a bitmap. From the fuzzer's perspective, the second input deserves no attention, as the first entry had already set the corresponding bitmap entries, meaning the second input has not explored new program behavior. We say these two inputs share a *novelty equivalence class*, with a more formal definition being:

Definition 1: Novelty Equivalence Class

For a given program p with the input space I and a fuzzer f, a *novelty equivalence class* v comprises all possible inputs $i \in I$ that p accepts and that, when executed, yield the same coverage measurement to f.

A difference between inputs in this novelty equivalence class is solely discernible on the byte level (crucially, without this difference impacting the execution path). For the sake of an example, suppose we have two inputs i_1 and i_2 that share a novelty equivalence class v. i_2 is not necessarily better or worse than i_1, but the crucial point is that the difference on the byte-level allows the fuzzer's mutations to create other inputs from i_2 than from i_1. Consider an example program with the input space shown in Figure 1: Our mutations would be able to find another interesting input, t, when applied on i_2, however, not on i_1. As both i_1 and i_2 share the same coverage and i_1 is discovered first, the fuzzer will not consider input i_2 and, thus, not generate input t. Consequently, we may never exercise this part of the program, as the fuzzer blocks itself. We say that i_2 is *shadowed* by i_1, or more formally:

Definition 2: Input Shadowing

An input i_2 is *shadowed* by another input i_1 if both inputs share a novelty equivalence class v and i_1 has been executed before i_2, such that the execution of i_2 achieves no new coverage.

We emphasize that input shadowing is not necessarily bad. After all, it is a natural consequence of the fuzzer's novelty search that focuses on discovering new program behaviors rather than re-testing old ones. That said, we believe the downsides of input shadowing to be largely overlooked in the literature: A shadowed input i_2 may be a necessary predecessor to some bug-inducing behavior, meaning the fuzzer effectively locks itself out by discarding i_2.

We illustrate this behavior using the exemplary function in Algorithm 1 as fuzz target. Our target function takes two parameters: an unsigned 32-bit integer a and an unsigned 16-bit integer b. The function's purpose is to check if the sum of a and b is less than 128, provided that a is smaller than 256. If both conditions are met, execution is aborted. Consider a fuzzer that generates inputs for this function by randomly selecting values from a uniform and independently distributed range of 32-bit integers. Suppose the fuzzer initially chooses $a = 200$. It successfully solves the first branch and records the feedback in its bitmap. In the subsequent step, the fuzzer attempts to satisfy the next branch condition ($a + b < 128$) by mutating the variable b while maintaining the previously discovered value for a. However, with the value of 200 for a, it becomes impossible to fulfill the second branch condition. If the fuzzer now mutates the value for a again, opting for $a = 50$ and $b = 100$, the first condition would be satisfied once more but not the second one. However, due to the novelty search, the fuzzer does not save the new value for a because the fuzzer has already identified an input achieving the exact same coverage ($a = 200$). In other words, the first input ($a = 200$) *shadows* the second one ($a = 50$). Nevertheless, it is easy to see that $a = 50$ would have been the better choice to meet the second condition. Now, this does not mean the fuzzer may never solve the branch, but it decreases the likelihood: The fuzzer can still correctly guess two new values for a and b in a single iteration, such that $a + b < 128$. However, this

Algorithm 1 Example function showcasing impact of input shadowing

```
function FOO($a : u32$, $b : u16$)
    if $a < 256$ then
        if $a + b < 128$ then
            ABORT
        end if
    end if
end function
```

is less likely if the fuzzer only had to guess the value for b. While this example is simplistic and artificial, it showcases that in some situations the fuzzer can benefit from other inputs in the same novelty equivalence class.

Consequently, the question is whether we can mitigate the downsides of input shadowing without sacrificing the benefits of the novelty search that is one of the driving factors behind modern fuzzer success. A naive way to mitigate the impacts of this phenomenon is to reset the fuzzer state (in terms of saved inputs and the bitmap) and observe how a new, "clean" run performs, which we can achieve by restarting the fuzzer. Intuitively, this sounds like a bad idea: After all, we already spent significant resources in terms of executions for driving the exploration of the program towards new, unseen behaviors.

2.3 Basic Block Frequencies

To analyze the impact of such state-resetting restarts, we conducted a fuzzing campaign on `libpng` and recorded the basic block hits throughout the fuzzing runs. More precisely, we run AFL++ multiple times for 24 hours using different tweaks: As a baseline, we use a standard setup of AFL++ without any restarts. Additionally, we use five setups with restart frequencies of 12, 6, 2, and 1 hours, as well as 15 minutes, resulting in 1, 3, 11, 23, and 96 restarts, respectively. To account for inherent randomness, we conduct the experiment ten times for each fuzzer and use the median trial for the subsequent analysis. The results are shown in Table 1 and Figure 2. Next, we discuss this experiment in more detail.

Contrary to the intuitive assumption that discarding state will lead to a lower coverage, we find that all runs using restarts have slightly higher branch coverage than the baseline (see Table 1). Even more interestingly, we find that restarted runs show a more diverse exploration behavior: Across the board, basic blocks are hit more often compared to the baseline. To study this phenomenon in detail, we sample every 10,000-th input during the runs and record the basic blocks it reaches. In this process, each basic block is counted only once per sample. Therefore, we did not count the number of hits for each basic block, but rather the number of samples that reached each basic block. Importantly, we do

Table 1. Branch coverage from `libpng` of the median run from ten 24 hour runs and basic block hits that are *more than twice* as often triggered by the baseline 🟦 or our restarted fuzzer 🟥.

		#Basic Blocks favoring	
	Median #Branches	🟦 Baseline	🟥 Tweak
AFL++	2006	–	–
`no-restart`	2006	426	376
`reset-` 720min	2011	146	542
`reset-` 360min	2015	122	597
`reset-` 120min	2016	95	640
`reset-` 60min	2017	104	642
`reset-` 15min	**2019**	89	**717**

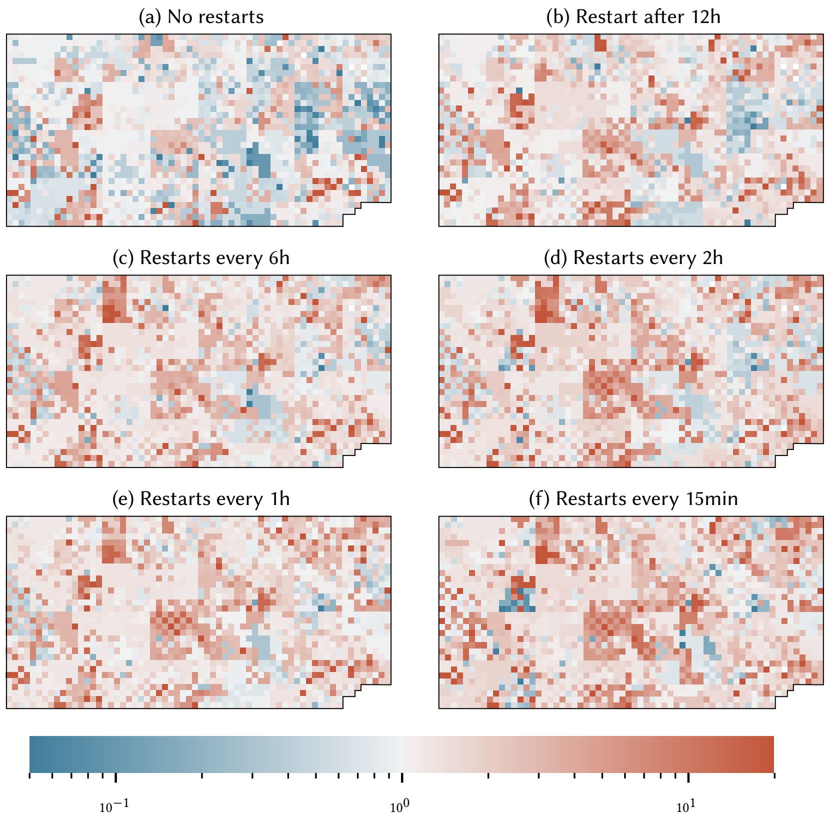


Fig. 2. Heatmaps based on Hilbert curves showing the frequency in logarithmic scale of basic block hits during a fuzzing run relative to a baseline run for the target `libpng` (we use the median run from ten 24 hour runs for each). Each cell represents a basic block; blue blocks 🟦 indicate the median baseline run hit this basic block more frequently, and red 🟥 indicates the opposite. This shows that fuzzer restarts diversify the basic blocks visited, as its inherent randomness causes the fuzzer to hit different basic blocks.

not work on the *queue* to avoid survivorship bias: As each input accepted into the corpus is novel, if we sampled from the queue, we would blur the experiment's results, biasing our analysis towards such inputs while ignoring shadowed inputs. Due to the large number of inputs executed by fuzzers, we only sample the covered basic blocks for a subset of them. We show the results of this experiment as heatmaps, transforming the 1-d coverage bitmap to a 2-d representation using a space-filling Hilbert curve in Figure 2, comparing the frequency of basic block hits against the baseline without any restarts. Each cell marks a distinct basic block; blue 🟦 indicates the basic block was hit more often by the baseline, while red 🟥 denotes the basic block was visited more often by the tweak. White indicates equal frequency, meaning both fuzzers visited the basic block the same number of times. The difference is represented through the shade, with darker colors representing a larger difference. Without restarts, rare edges are hit less often, if at all. Instead, non-restarting runs hit the more frequent blocks even more often than restarting ones. As we use a logarithmic scale, this difference appears less pronounced in the plots.

When running this experiment, one may intuitively assume that fuzzing runs with roughly equal coverage would exercise edges at approximately the same frequency. However, when we compare two ordinary fuzzing runs that were not restarted, we find the opposite: As depicted in Figure 2a, the distribution of frequencies shows differences exceeding an order of magnitude. In other words, each fuzzing run stresses different parts of the program while still achieving comparable code coverage. This implies that the final coverage results (in the form of the corpus) are no accurate representation of the input distribution tested during the fuzzing run. Executing a novel coverage once suffices to add this input to the corpus, even if it is never tested again – similarly, a basic block exercised thousands of times may be represented by a single input in the corpus. We emphasize that this observation does not imply that measuring code coverage is an inferior metric, as it is still an excellent proxy for the observed program states.

Our initial results suggest that simply running the fuzzer helps to visit basic blocks more often, without sacrificing overall exploration. Speculatively, exploring basic blocks with different inputs may help to uncover bugs [18]. However, one problem with running the fuzzer multiple times is the associated cost: Running it a second time effectively doubles the cost, rendering the benefit of potentially having some different inputs tested unclear. To avoid this cost, we could restart the fuzzer after half the runtime, say, after 12 hours. This would probably lead to a better frequency distribution of basic block hits, but at the cost of potentially missing some of the coverage. To investigate whether this works in practice, we restarted the fuzzer in different frequencies, ranging from one time (after 12 hours) up to 96 times (every 15 minutes) resulting in Figures 2b, 2c, 2d, 2e, and 2f, respectively. As these heatmaps show, a higher number of restarts is indeed helpful for diversifying the set of hit basic blocks. Interestingly, even a single restart after 12 hours (Figure 2b) increases the block frequency. However, restarting the fuzzer more than once shifts the frequency distribution clearly in favor of the tweak, i. e., the restarted fuzzer. At the same time, the more restarts occurred, the more coverage was found, even though it is still similar to the baseline.

In summary, our experiment indicates that fuzzer restarts can potentially help to increase the basic block hit frequency and therefore help to trigger rare blocks more frequently. We hypothesize that this may help the fuzzer's bug finding capability. At the same time, restarts have a positive but small impact on the coverage found. We speculate that with a better, more adaptive strategy than restarting the fuzzer after a fixed period of time, we can further increase the benefits.

2.4 Challenges

A better strategy to overcome these fixed interval restarts must mainly solve two challenges. First, we must determine a "good" point in time when to reset the fuzzer. Restarting the fuzzer too early will prevent it from exploring deeper program parts meaningfully. Restarting the fuzzer too late, on the other hand, spends fuzzing time without accounting for the effect of input shadowing. Second, fully resetting the fuzzer state discards valuable information. We hypothesize that maintaining a balance between keeping valuable information, such as seeds unlocking new regions of the program under test, and discarding this information that potentially shadows other inputs could be worthwhile. The underlying insight is that solving some fuzzing roadblocks and uncovering new coverage can be challenging, potentially restricting the fuzzer to a small part of the program when continuously restarting it. The challenge is to find an optimal balance between information kept and discarded. Ultimately, the ideal strategy would be to restart the fuzzer when the likelihood of finding new coverage is below a certain threshold and discard all inputs leading to this situation. In practice, the lack of information renders both challenging. While we could collect more precise information during fuzzer runtime, this increases the runtime overhead, reducing the fuzzer's effectiveness.

3 ADAPTIVE FUZZER RESTARTS

To overcome these challenges and mitigate input shadowing without sacrificing code coverage, we propose a scheduler that uses several techniques to reset the fuzzer's state at the right moment.

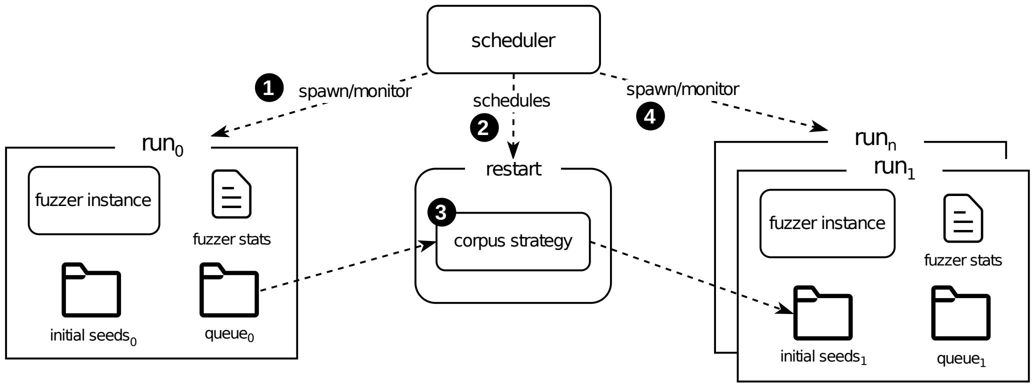


Fig. 3. Overview of our restart scheduler

3.1 Restart Scheduler Overview

The restart scheduler is responsible for the orchestration of a fuzzing campaign. It is not to be confused with fuzzer-internal seed scheduling. In our case, the scheduler's task is to monitor a fuzzing run and restart the fuzzer according to some metric discussed subsequently. This restart may include the preservation of the fuzzing state.

To explain the scheduler's behavior across a fuzzing campaign, we show an overview in Figure 3. First, an initial run_0 is launched by spawning a fuzzer instance. As with traditional fuzzing, we need to provide it with a set of initial seeds. Then, our scheduler monitors this instance during fuzzing ❶. In particular, it tracks changes, such as paths found, which are used to decide whether to restart ❷ the fuzzer based on some restart policy. In the event of a restart, the currently running fuzzer instance run_{n-1} is stopped, and—if we want to keep some state across the restart—its resulting *queue* is processed according to our corpus strategy ❸. The preserved state as a subset of the *queue* is then passed to the next fuzzing run run_n as initial seeds (alongside the original seeds). The process then continues with monitoring the new instance ❹ until the next restart is due.

3.2 When to Restart: Coverage Plateaus

The first challenge is identifying a suitable time to restart the fuzzer.

Coverage plateaus. We rely on the insight that all fuzzing runs at some point hit so-called *coverage plateaus*. This denotes the time when the fuzzer, despite continuously mutating inputs, fails to uncover any new coverage. This manifests as a plateau in coverage plots, yielding the descriptive name. Retrospectively, such coverage plateaus are easily identified; during the actual run, it is difficult to predict whether the fuzzer is stuck in a plateau or is close to solving some complex constraints that yield new coverage. To achieve a computationally feasible yet effective recognition of such plateaus, we propose the following heuristic: We consider the fuzzer to be situated in a plateau when it fails to uncover new coverage, i. e., new edges of the program, for n minutes, where n is a small number such as 15. As the experiment from our motivation shows, small restart times have the greatest benefit for a better distribution of the basic block hits.

Detection heuristics. Our scheduler features several strategies for deciding on the point in time at which to restart the fuzzer. The first and most naïve one does not rely on detecting coverage plateaus but instead uses a fixed or random countdown and restarts the fuzzer once this countdown expires (akin to the strategy used in the motivating experiment in Section 2). While this heuristic is straightforward, it has a major drawback: It does not account for the current performance of the fuzzing run. Using this heuristic might cause the fuzzer to be restarted while discovering a new region of the target program that has not been covered yet. To avoid degrading the fuzzer's performance because of poorly timed restarts, our restart scheduler monitors the coverage progress of the fuzzer to determine whether the fuzzer is currently advancing. Every time the fuzzer finds new coverage, the countdown is reset, essentially only restarting the fuzzer when it hits a coverage plateau and fails to find new coverage for a certain amount of time. This *adapts* the fuzzer restarts to the program under test. The combination with small restart times and the coverage plateau detection allows the scheduler to perform more restarts on targets that reach a coverage plateau early and extend the restart times on targets where the fuzzer is able to exercise new coverage.

Unfortunately, this approach still has a disadvantage for targets where new coverage is found slowly but steadily. Imagine a fuzzer that finds a single new edge every five minutes: We speculate that restarting aggressively despite finding this edge can be beneficial. Thus, our scheduler features a third heuristic that uses a threshold allowing the configuration of the degree of coverage growth that causes our restart timer to be reset. The threshold is calculated over the fuzzing progress observed in the past and prevents the scheduler from being stalled by programs with negligible coverage growth over time.

In summary, our restart scheduler features the following three heuristics:

(1) Blind restarts with fixed or random reset countdown
(2) Coverage plateau-based restarts
(3) Coverage growth rate-based restarts using a threshold

3.3 How to Restart: Corpus Retention

The second crucial design decision we must consider for our scheduler is *how much* of the fuzzer's state should be reset. This directly translates to how much of the generated corpus we want to preserve across fuzzing runs by passing them as initial seeds to the subsequently started run.

Motivation. The most basic approach is to reset the complete state of the fuzzer and start in a new environment, such that the new instance has no information about the previous advancement. This approach has the disadvantage that the new fuzzing run must re-explore the entire program. For this reason, more advanced strategies are desirable, which keep parts of the generated corpus from the previous run to consecutively transfer some of the coverage information to subsequent runs. This partly preserved corpus bootstraps the fuzzer, since it immediately has several inputs that explore different program behaviors.

Corpus retention strategies. Our scheduler is equipped with the following six strategies to identify the parts of the corpus to keep.

(i) Reset. This strategy represents a *full* reset; it does not keep any files from the corpus, thus not preserving any information.

(ii) Input Shuffle. The `input shuffle` strategy is the opposite of our `reset` strategy. When a restart occurs, all corpus files are shuffled by assigning a random numerical prefix to each testcase name. This leads to a different order in which the corpus files are read by AFL++. The renamed files are then used as a new seed corpus for the next run; therefore nothing is discarded.

(iii) Corpus Pruning. Upon a restart, `corpus pruning` randomly selects a percentage between 5% and 95% of generated inputs from the corpus to delete. Removing more inputs reduces the effects of input shadowing, while at the same time requiring the fuzzer to spend more cycles rediscovering inputs. We speculate there is no universally good formula to decide which inputs to discard, as it is a highly target-dependent problem. Thus, our scheduler uses random choice, in the spirit of fuzzing, to explore both directions in a balanced way.

(iv) Time Travel. The `time travel` strategy selects a random timestamp at which the corpus of the current run is a snapshot. Upon restart, our scheduler passes this snapshot of the corpus to the new fuzzing run. This effectively deletes all files in the corpus found after a specific point in time. The motivation is to reset the fuzzer's attention and allow it to re-explore the program from this specific point in time.

(v) Tree Chopper. The idea behind the `tree chopper` strategy is to remove all offspring from one or more selected files within the generated corpus. This allows us to identify trees of inputs, i. e., one input and all subsequent inputs derived from this one via mutations. The scheduler selects a random subtree in the input-mutation graph; then, all offspring originating from this input are removed from the corpus. This approach allows the fuzzer to explore different states based on the remaining corpus and potentially make alternative decisions regarding the remaining corpus.

(vi) Tree Planter. The `tree planter` strategy is similar to `tree chopper`, with the key difference being that all generated trees except a single one are removed. This may allow the fuzzer to explore a particular part of the target in greater depth.

(vii) Ensemble. Similar to ensemble fuzzing, where multiple fuzzers are combined and scheduled adaptively [13, 20, 26], we hypothesize that an ensemble strategy, which dynamically selects the most suitable retention strategy for the respective target under test, could improve the efficiency. Thus, we propose an ensemble strategy that leverages a variety of retention strategies throughout the fuzzing process to adapt to the evolving characteristics of the target application.

In detail, our ensemble strategy systematically cycles through different retention strategies, initiating the fuzzing campaign with the `reset` strategy to establish a starting point for subsequent runs. In the following runs, a retention strategy is then randomly selected from the available options (prioritizing never tested ones). More precisely, the scheduler evaluates the performance of a chosen strategy over the next three restarts. Once the fourth restart occurs, the scheduler decides whether to retain or replace the current strategy for the next runs based on its performance. More precisely, it compares the average *corpus counts* for the current strategy with the overall average from the previous runs. Beyond the performance of the currently active strategy and the average performance observed across all tested strategies, the scheduler also keeps a record of the best-performing strategy it has observed across all restarts, ensuring it can always fall back to the best strategy observed when others prove unsuccessful.In the case this best-performing strategy no longer gains new coverage over multiple restarts, the scheduler will replace it with a random one.

Given that the ensemble strategy necessitates multiple runs, it is particularly well-suited for long-term fuzzing campaigns and targets that exhibit coverage plateaus after an initial exploration period. By dynamically adapting to the changing dynamics of the target application, the ensemble strategy aims to maximize the overall code coverage and vulnerability discovery potential over extended fuzzing campaigns.

4 IMPLEMENTATION

To test the impact of restarts during a fuzzing campaign, we implemented our scheduler in a prototype called `Sileo`, which spawns and monitors the underlying fuzzer instances. Our prototype is written in 1, 358 lines of Python code and uses AFL++ in version *4.04c* for fuzzing the target under test. `Sileo` can be easily combined with other fuzzers, as it requires only three primitives: (1) `Sileo` must receive information regarding the coverage achieved so far (e. g., the number of edges found). (2) It must have access to the inputs considered interesting (i. e., the queue) to apply one of the corpus retention strategies. (3) `Sileo` must be able to start and stop a new fuzzer instance. As these requirements are satisfied by virtually every general-purpose fuzzer, replacing `Sileo`'s underlying fuzzing engine is straightforward.

In the case of AFL++, the coverage can be monitored by observing the `fuzzer_stats` file, which periodically exports several fuzzer metrics via the filesystem. Likewise, access to the generated inputs can be achieved by reading the content of AFL++'s `queue` directory. Since AFL++ is a user-space application, the spawning and termination of a fuzzer instance can be facilitated using standard OS primitives for process creation and termination.

Coverage plateau detection heuristics. To detect coverage metrics, we periodically poll the `fuzzer_stats` file and examine the elapsed time since the `last_find` and the `corpus_count`. A restart is triggered if the `last_find` time exceeds the restart countdown. When the threshold-based restart heuristic is used, we save the current `corpus_count` and calculate the threshold based on the last n values of these corpus counts. If the restart countdown has elapsed and the threshold has been reached, a restart is triggered.

Corpus retention strategies. We implement the different corpus retention strategies introduced in Section 3.3 in `Sileo`. After a restart has been scheduled, first, a copy of the current queue is generated. After that, depending on the chosen retention strategy, some files are potentially discarded and shuffled to account for input shadowing, while importing the newly generated seed corpus.

For our experiments, we used FuzzBench [42]. However, we observed that disk space usage increases significantly when using our corpus-based strategies, since FuzzBench creates periodic snapshots of the corpus. To mitigate this issue, we added an option to FuzzBench so it only stores the last two corpus archives. Additionally, to facilitate sampling of every input (instead of only those in the corpus), we patch AFL++ and add functionality to store every n-th execution. This feature is used during evaluation to compute metrics, like the heatmaps from Figure 2 and our basic block frequency plots, over the generated fuzzer test cases.

5 EVALUATION

For our evaluation, we use our prototype `Sileo` and test it with different heuristics to determine when to restart the fuzzer as well as the efficacy of various corpus retention strategies.

During the course of our evaluation, we answer the following three research questions:

- **RQ1:** Are fuzzer restarts an effective and efficient mitigation of the downsides of input shadowing?
- **RQ2:** What is the impact of fuzzer restarts on code coverage?
- **RQ3:** Can fuzzer restarts help find bugs in software?

Experiment Setup. We conducted our evaluation on a server equipped with an Intel Xeon Gold 6230R CPU featuring 52 cores at 2.10GHz and 196GB of memory, running Ubuntu 22.04. Using FuzzBench, we orchestrated the fuzzing campaign for the coverage experiments, considering each strategy as an individual fuzzer. All fuzzers were run for 24 hours, and each experiment was repeated ten times to account for inherent randomness in the fuzzing process, following the recommendation by Klees et al. [32] and Schloegel et al. [54]. In addition to our coverage experiments, we executed a long-term run of one target and bug-finding experiments. To preempt external factors from interfering with our experiments, we assign each run of every fuzzer to a dedicated physical core and disable the CPU's turbo-boost feature, such that it maintains a consistent frequency for each core, even in case of partial CPU utilization.

Fuzzbench and Target Applications. The majority of our experiments are conducted using the benchmark framework FuzzBench [42][1]. We employed a custom version of FuzzBench with modifications to the code base that reduce overhead in terms of disk space and address minor bugs. We will add our version of FuzzBench to our artifact.

FuzzBench offers a diverse set of target applications to assess fuzzer performance in terms of coverage and bug discovery. We selected a total of 15 targets, comprising twelve FuzzBench targets[2] and one additional target outside of FuzzBench (`objdump` 2.41.50.20231023). In addition, eight further FuzzBench targets are designated for bug-finding experiments. Our choice of FuzzBench targets was informed by official public reports, demonstrating their variability in coverage across different fuzzers.

Selected Restart Strategies and Heuristics. Our prototype `Sileo` features three different restart heuristics and six retention strategies. Since the plateau-based heuristic leads to few (if any) restarts on some targets, we chose the coverage growth heuristic for all our experiments to decide when to restart the fuzzer. From our six corpus retention strategies—that decide what parts of the generated corpus are used as seeds after a restart—we select a subset of three strategies (`reset`, `corpus pruning`, and `input shuffle`) for the subsequently presented evaluation to maintain readability. Interested readers can explore our artifact[3] to find *all* proposed restart strategies (`tree planter`, `tree chopper`, `time travel`) for all tested FuzzBench targets. We chose the named three strategies, as `reset` and `input shuffle` represent excellent baselines for `Sileo` itself, and `corpus pruning` has proven to be the best-performing strategy. `reset` resets the complete fuzzer state, not using any part of the generated corpus for the next run, while `input shuffle` retains the entire corpus and shuffles all files randomly, allowing the fuzzer to start with a vast seed corpus in the next run. Positioned as a middle ground between these two strategies, `corpus pruning` is then employed.

Fuzzers and Baselines. We use AFL++ in version 4.04c [19] as the foundation for `Sileo` and, consequently, as a baseline without restarts. Due to our various restart strategies, we build each strategy on top of AFL++ and add it as an individual fuzzer to FuzzBench, facilitating comparability with our baseline and other strategies.

Coverage Computation. We use FuzzBench, which internally uses `llvm-lcov`, to compute coverage. FuzzBench generates a detailed report that not only illustrates the found coverage of the conducted experiment but also provides additional information, such as a statistical evaluation, to confirm the statistical relevance of the results. For our target `objdump`, which we executed without FuzzBench, we also used `llvm-lcov` to calculate coverage metrics. To do this, we executed all generated fuzzer inputs for a target from all fuzzers on the same coverage binary to ensure consistency and comparability of results. This approach enables a standardized assessment of code coverage across different fuzzing strategies, contributing to the robustness and reliability of our experimental evaluations. d

5.1 Experiments

To evaluate the effectiveness of restarting the fuzzer, we compare our various restart strategies against AFL++, using the selected target applications (15 in total). Besides the experiment with our different restart strategies, we also perform

[1]commit: 6ca67572eed7035fe7fb0655d5d40ad426c92a3d
[2]We used the OSS-Fuzz [43] benchmark integration script provided by FuzzBench to integrate targets beyond the default ones
[3]https://github.com/CISPA-SysSec/fuzzing-restarts

an experiment that uses *afl-cmin* and a long-term experiment over 7 days with the `ensemble` strategy. Each target was run for 24 hours (except the long-term experiment) with ten repetitions, and we plot the median over all runs and the 60% trimmed interval to omit outliers. For brevity, this paper presents the results for a subset of our targets, while the complete set of results for all targets is available in our research artifact available online. In summary, we conducted the following six experiments to address our research questions:

(1) Main coverage experiment spanning a total of nine targets
(2) Coverage experiment investigating the effect of corpus minimization after restarts
(3) Long-term coverage experiment (runtime of 7 days per fuzzer) that uses our `ensemble` strategy
(4) Coverage experiment studying fixed restart times
(5) Sampling experiment on the targets `sqlite3`, `libpcap`, and `libpng`
(6) Bug experiment on eight FuzzBench targets

Experiment 1: Coverage Experiment. We show the results of our main coverage experiment in Figure 4 and various statistics in Table 2. Overall, our results indicate that restarting the fuzzer positively impacts code coverage across all selected targets, with at least one of the restart strategies outperforming the baseline AFL++ in terms of covered branches. This observation aligns with the findings from the full coverage report on all nine targets from the research artifact. When comparing the performance of different restart strategies, it becomes evident that their effectiveness heavily depends on the target itself: In three out of the six targets, `corpus pruning` performs best, especially on `objdump`, where it achieves significantly more coverage than both `reset` and `input shuffle` as well slightly outperforming AFL++. We also observe a similar performance for `sqlite3`, where `corpus pruning` outperforms AFL++ and `input shuffle`. When comparing `reset` and `input shuffle`, `reset`'s median of covered branches outperforms `input shuffle` in four out of six targets (see Figure 4 and Table 2), whereas in two of these four targets, the median lies within the trimmed 60% intervals.

To confirm our results statistically, we follow the recommendations of Arcuri and Briand [2] and consider the two-sided non-parametric Mann-Whitney-U test as well as the effect size as determined by Vargha's and Delaney's $\hat{A}_{12}$ test [61]. We show them in Table 2.

For `corpus pruning`, all results except for `mupdf` compared to the baseline are statistically significant, and the effect sizes are large. For `mupdf`, the effect size is medium, but the difference is statistically insignificant. Regarding our `input shuffle` strategy, only two cases show statistical significance (`bloatyfuzz` and `objdump`), with both cases having a large effect size, but on the target `objdump`, the observed effect size is negative. Our last strategy, `reset`, exhibits statistical significance for all six listed targets with a large effect size. However, the effect size for two out of the six targets is negative (`objdump` and `sqlite3`), indicating that `reset` is outperformed by AFL++.

To analyze our techniques, we track the number of restarts performed by `corpus pruning`, `reset`, and `input shuffle` throughout the evaluation, as summarized in Table 2. To illustrate the actual impact of a restart on coverage, we plot the median run of four targets in Figure 5 and indicate the position of a restart using the respective marker symbol. As depicted, a restart may not necessarily result in an immediate jump in new coverage; instead, it can lead to a slow but steady growth effect. The data reveals that `corpus pruning` requires significantly fewer restarts (up to 13 times less, with an average of 12 restarts across all targets) compared to `input shuffle` (with an average of 61 restarts across all targets), while still achieving higher coverage in five out of six targets. When comparing the number of restarts between `corpus pruning` and `reset`, the latter also performs more restarts (with an average of 25 restarts across all targets), although it outperforms `corpus pruning` only on two out of six targets slightly. Upon comparing

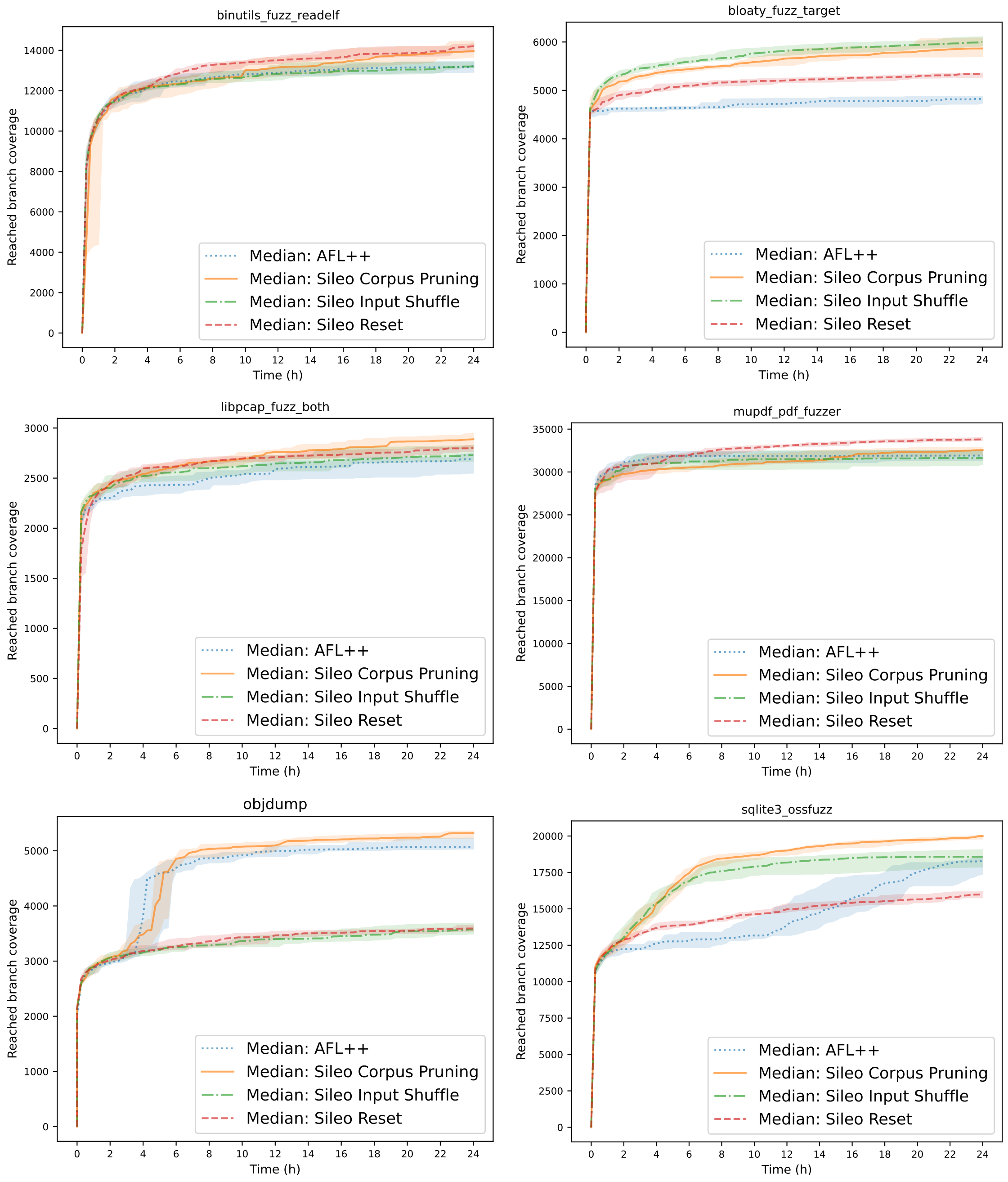


Fig. 4. The coverage (in branches) for different tools of 24 hour runs. Displayed is the median coverage over ten repetitions, and the trimmed 60% interval.

the number of restarts with the achieved branch coverage and the respective corpus retention strategy, it becomes evident that the success of our approach is not solely determined by the frequency of restarts. Rather, it heavily relies on the interplay between the corpus retention strategy employed and the specific target under test. A detailed examination of the restart frequency in relation to corpus preservation reveals that having a high number of restarts or preserving the entire corpus does not necessarily entail an advantage. The results suggest that dynamically preserving specific parts of the corpus leads to a lower number of restarts. All three strategies share the same restart heuristic (coverage growth), and our findings indicate the following:

- *i)* Preserving the entire corpus often results in a saturated corpus after a short amount of time. Utilizing such a saturated corpus as the initial seed corpus for the fuzzer makes finding new inputs that explore new coverage challenging. Consequently, coverage growth stagnates quickly, triggering another restart. While a restart offers the advantage of a complete reset of the fuzzer bitmap, it still suffers from corpus saturation.
- *ii)* Preserving nothing from the corpus and discarding the fuzzer progress entirely generally leads to more restarts. The issue here is that the "first" restart (after starting "fresh") occurs more frequently due to always having the same set of initial seed files, making it more challenging to reach deeper code paths before a new restart. Similar to (i), a restart has the advantage of a complete reset of the fuzzer bitmap but suffers from having the same set of seeds in the next run, providing the fuzzer with the same starting point.

As a compromise between `input shuffle` and `reset`, it is unsurprising that `corpus pruning` achieves the best performance across almost all targets. However, despite these negative effects, we observe that both `reset` and `input shuffle` perform almost as good as or even slightly better than `corpus pruning` on specific targets (e.g., applying `reset` on mupdf). After studying these targets, we believe these to be cases where the benefits of restarting more often

Table 2. Statistical analysis of our code coverage experiment, with the hypothesis that **tweak** performs better than **baseline**. We use the median covered branch values to measure the effect size using Vargha's and Delaney's $\hat{A}_{12}$ test and use their categorization (L = large, M = medium, S = small; a minus represents a negative effect size, indicating the tweak is worse than the baseline) [61]. We use the two-sided non-parametric Mann-Whitney-U test and mark $p < 0.05$ in bold.

Tweak	Baseline	Target	#Restarts	Tweak	Baseline	Effect Size	p-value
`corpus pruning`	AFL++	`readelf`	10	**14,084**	13,379	+L(0.82)	**0.0147**
		`bloatyfuzz`	32	**5,902**	4,842	+L(1.00)	< **0.0001**
		`libpcap`	12	**2,891**	2,693	+L(0.93)	**0.0005**
		`mupdf`	6	**32,642**	32,179	+M(0.71)	0.1230
		`objdump`	8	**5,330**	5,073	+L(0.87)	**0.0039**
		`sqlite3`	4	**20,006**	18,328	+L(0.99)	< **0.0001**
`input shuffle`	AFL++	`readelf`	62	13,201	**13,379**	(0.52)	0.9118
		`bloatyfuzz`	81	**6,053**	4,842	+L(1.00)	< **0.0001**
		`libpcap`	72	**2,739**	2,639	+S(0.64)	0.3150
		`mupdf`	14	31,699	**32,179**	(0.46)	0.7959
		`objdump`	82	3,567	**5,073**	-L(0.00)	< **0.0001**
		`sqlite3`	53	**18,615**	18,328	+S(0.62)	0.3930
`reset`	AFL++	`readelf`	13	**14,233**	13,379	+L(0.97)	< **0.0001**
		`bloatyfuzz`	71	**5,342**	4,842	+L(1.00)	< **0.0001**
		`libpcap`	19	**2,803**	2,639	+L(0.87)	**0.0039**
		`mupdf`	17	**33,854**	32,179	+L(0.99)	< **0.0001**
		`objdump`	12	3,599	**5,073**	-L(0.00)	< **0.0001**
		`sqlite3`	19	16,006	**18,328**	-L(0.19)	**0.0185**

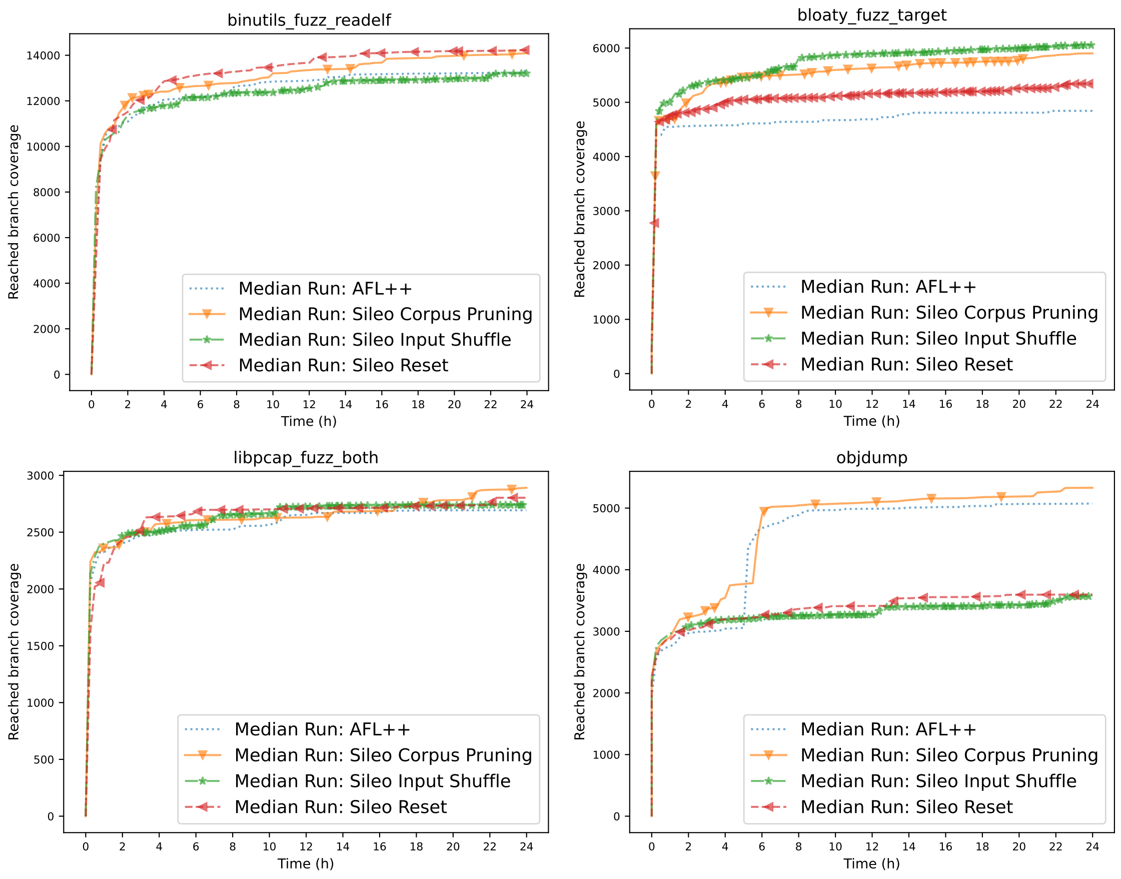


Fig. 5. The coverage (in branches) for four different tools of 24 hour runs. Displayed is the median run of ten repetitions and for `Sileo` the restarts are marked. For all plots of all tested tools, have a look at our artifact on GitHub.

outweigh the disadvantages outlined in *i*) and *ii*). Overall, these results show that restarting the fuzzer during a fuzzing campaign has indeed a positive impact in terms of coverage and therefore allow us to answer our **RQ2** positively.

Experiment 2: Corpus Minimization after Corpus Retention. Depending on the target, our corpus retention strategies often generate large amounts of data, as with each restart, inputs previously found by the fuzzer can be rediscovered (we still need to store the previous queue for coverage computation). This results in ever-increasing disk usage for the corpus in each run. In addition to the growing disk usage, a large seed corpus has the disadvantage that AFL++ has to process all these files during its startup calibration phase. Therefore, a more compact corpus appears preferable for the next run after a restart occurs. AFL++ provides a script called `afl-cmin`, which can be used to *minimize* the corpus. The script uses `AWK` and `afl-showmap` to reduce a corpus of input files. Its primary function is to identify and retain the smallest set of unique test cases that collectively cover the diverse code paths explored by the original corpus. By analyzing the instrumentation output generated by AFL++ using `afl-showmap`, it prioritizes the retention of input files that contribute distinct code coverage, aiming to reduce redundancy.

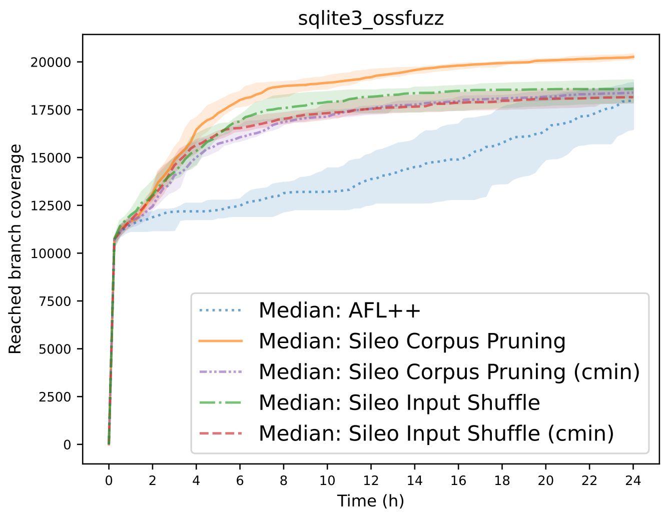
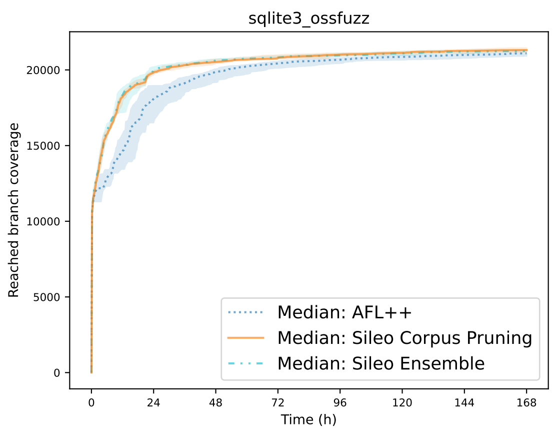


Fig. 6. The effect of `afl-cmin` after every restart to minimize the seed corpus for the next run. Displayed is the median coverage over ten repetitions and the 60% trimmed interval.

We incorporated this functionality into our scheduler to investigate the effect of corpus minimization after a restart and conducted an experiment using our `corpus pruning` and `input shuffle` strategy with `afl-cmin`. Note that due to a bug in AFL++ 4.04c, we changed the version of all fuzzers for this experiment to the newer version 4.06c. When a restart is triggered, `afl-cmin` performs the corpus minimization, and the remaining files are then used as seeds for the next run.

In Figure 6, the performance of AFL++ is compared with `corpus pruning` and `input shuffle` to the respective versions that are using `cmin`. The plot indicates that the usage of `afl-cmin` *negatively* affects the performance of `corpus pruning`, leading to an earlier flattening of the coverage growth. In contrast, the impact of `cmin` to `input shuffle` is minimal. Interestingly, the `cmin` strategies perform almost as good as their counterparts but then flatten earlier. The behavior observed in the experiment is expected when considering input shadowing, where the minimization process (focusing solely on code coverage) may discard inputs that contribute to deeper code exploration. Consequently, the fuzzer's ability to discover novel paths is slightly hindered, leading to less coverage achieved over time.

Experiment 3: Long-Term Run and Ensemble. We conducted a 7-day run on `sqlite3` to assess if AFL++ can catch up with `Sileo` over an extended period, as the 24 hour run indicated that AFL++ was closing in on `Sileo` over time. Additionally, such a long run allows us to test our `ensemble` strategy, which requires multiple restarts to identify and exploit the best-performing strategy. As shown in Figure 7, AFL++ catches up with `corpus pruning` and `ensemble`, but the median never reaches the same coverage as the two `Sileo` strategies. When comparing `corpus pruning` and `ensemble`, both perform equally over the entire fuzzing time. This is somewhat expected, as `ensemble` chooses its modes from our corpus retention strategies, and none of these modes performed better than `corpus pruning`, `time travel`, or `tree planter` on this target. Therefore, `ensemble` can only perform as well as the best mode for this target. This behavior is also evident in our log files, showing that `ensemble` mainly switches between these three modes. The



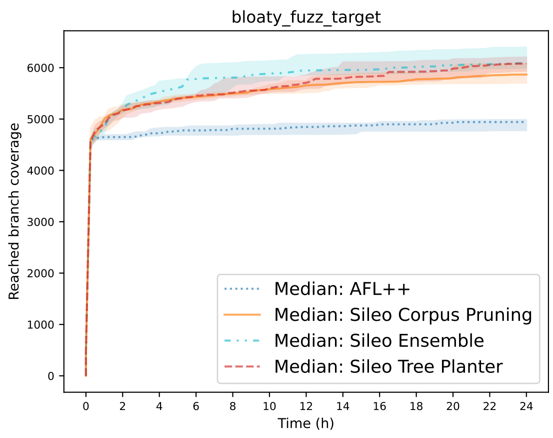


Fig. 7. A 7-day run on the target `sqlite3`, as well as another run of `bloatyfuzz` featuring AFL++, `corpus pruning`, and `ensemble`.

number of restarts for `ensemble` for this target ranges between 16 restarts (best mode: `tree planter`) and 89 restarts (best mode: `time travel`), with a median of 31 restarts with the `corpus pruning` mode.

We selected a second target, `bloatyfuzz`, to assess the performance of `ensemble` with a shorter runtime of 24 hours. This target experiences a high number of restarts within the first 24 hours for all our strategies, making it suitable for testing with `ensemble`. In this experiment, we plotted the best-performing single strategy (`tree planter`) alongside `corpus pruning` and `ensemble`. Examining the coverage plot in Figure 7 reveals that `ensemble` exhibits superior performance and wider variance in coverage compared to `corpus pruning` and `tree planter` within the first six hours. As the experiment progresses, `ensemble` maintains slightly higher variance overall than `tree planter` and `corpus pruning` but achieves roughly the same median coverage as `tree planter`.

Taking a closer look, we find that the number of restarts on this target ranges between 12 and 60, with a median of 18 restarts. When looking at the best performing strategy picked by `ensemble` after 24 hours (across 10 repetitions), we find the following results:

- 4/10 `input shuffle`, number of restarts: 13, 14, 58, 60
- 2/10 `tree planter`, number of restarts: 27, 30
- 2/10 `time travel`, number of restarts: 12, 19
- 2/10 `corpus pruning`, number of restarts: 13, 18

Interestingly, the best-performing individual strategy, `tree planter`, is not the most frequently used strategy in `ensemble` for this target; instead, `input shuffle` is the most frequently used. However, `ensemble` still achieves a higher median coverage than `input shuffle`, demonstrating the benefits of employing different strategies on one target to trigger synergy effects and potentially enhance overall performance.

Experiment 4: Fixed Restart Times and Restart Examination. To assess the effects of different restart times and to compare the effectiveness of our adaptively selected restart time with fixed restart times, we created different fuzzers for FuzzBench based on `corpus pruning`. These fuzzers use fixed restart times of 30 minutes, 60 minutes, and 240 minutes, respectively. We tested these strategies on all our targets and present a subset of the results in Figure 8. Please refer to our research artifact for the full data for all targets. Beyond achieved coverage, we study the growth of the corpus size for each strategy in this experiment (plots on the right side of Figure 8). Upon closer inspection of the coverage plots, it becomes evident that the fixed 30-minute restart time has a negative impact on covered branches for all three plotted targets. The longer restart times of 60 and 240 minutes perform slightly better than `corpus pruning` on two out of three targets. On the `sqlite3` target, both `corpus pruning` with adaptive restart time and `corpus pruning` with a fixed restart time of 240 minutes outperform the other strategies and the baseline AFL++.

Studying the corpus size plots, we can see that AFL++ features a continuous corpus growth over time. This matches our expectation that a fuzzer continuously adds new inputs to the queue, increasing its size over time. On the other hand, restarts discard parts of the corpus, leading to a decrease of the corpus size (as the queue of older runs is stored for coverage computation but not used by the current run and thus not included in the size computation). The fuzzers with shorter restart times cannot build up a large corpus for each run, since the frequent restarts remove parts of the corpus before it becomes larger. However, despite the corpus size being small compared to the two other strategies, restarting every 60 minutes incurs no disadvantage in terms of coverage on two out of three targets. We point out that tracking the corpus size solely visualizes how much information is discarded between runs, but it does not support a relationship to coverage, which we measure based on the queues of all restarted runs (rather than just the latest one).

Despite the good performance of `corpus pruning` with fixed restarts on these targets, `corpus pruning` with adaptive restarts remains the best-performing fuzzer across all targets. Overall, it demonstrates better flexibility and can better react to the target under test, triggering a balanced number of restarts. Another drawback associated with fixed restart times is the increasing overhead in terms of disk usage with every restart, especially with low restart times. For instance, having a fixed restart time of 60 minutes results in 23 restarts in 24 hours, only slightly improving coverage compared to restarting just seven times or less, but resulting in more than twice the disk usage.

Experiment 5: Sampling and Basic Block Frequency. Beyond code coverage, we analyze the benefits of fuzzing restarts in terms of better distributing the frequencies of hit basic blocks, i. e., whether the restarts spread the fuzzer's attention to a more diverse set of basic blocks. To this end, we use the inputs sampled during the fuzzing runs (similar to the experiment in Section 2). Importantly, we do not use the corpus to avoid biasing our results. To explore this behavior, we sampled a subset of all inputs executed by AFL++, `corpus pruning`, and `reset` on three targets: `libpng`, `libpcap`, and `sqlite3`. It is worth noting that, unlike the experiment in Section 2, the `Sileo` strategies here leverage the coverage growth heuristic to determine when to restart, instead of fixed restart times.

We report the median coverage values for the experiment in Table 3 and visualize the corresponding heatmaps in Figure 9. Additionally, we provide an alternative representation in Figure 10, illustrating the distribution of basic block hits as a Cumulative Distribution Function (CDF) for all three targets. For this representation, we study how often each edge is hit and sort them according to the number of hits (with the edge being hit most frequently being at $x = 0$ and the edge hit the least being the right-most one). For instance, the program's entry point, being the most commonly hit basic block, is situated at the far left side of the plot. The y-axis reflects the frequency of hits for all basic blocks. Note that the sorting is specific for each fuzzer: The edge at $x = 100$ may differ between AFL++ and `corpus pruning`, however, in both cases it asserts that there exist 99 edges that have been hit at least as often as this one.

Analyzing the CDF plots (Figure 10) reveals that, on all targets except `libpng`, `corpus pruning` triggers edges more frequently than the baseline, especially for edges that are visited less frequently. The CDFs demonstrate that `corpus pruning` and `reset` trigger most edges more often than AFL++ on the `libpng` target. This observation is also supported by the data in Table 3 and the visual representation in Figure 9. Both indicate that the `Sileo` strategies have an advantage over AFL++ in this regard. On `libpcap`, only `corpus pruning` performs slightly better than AFL++,

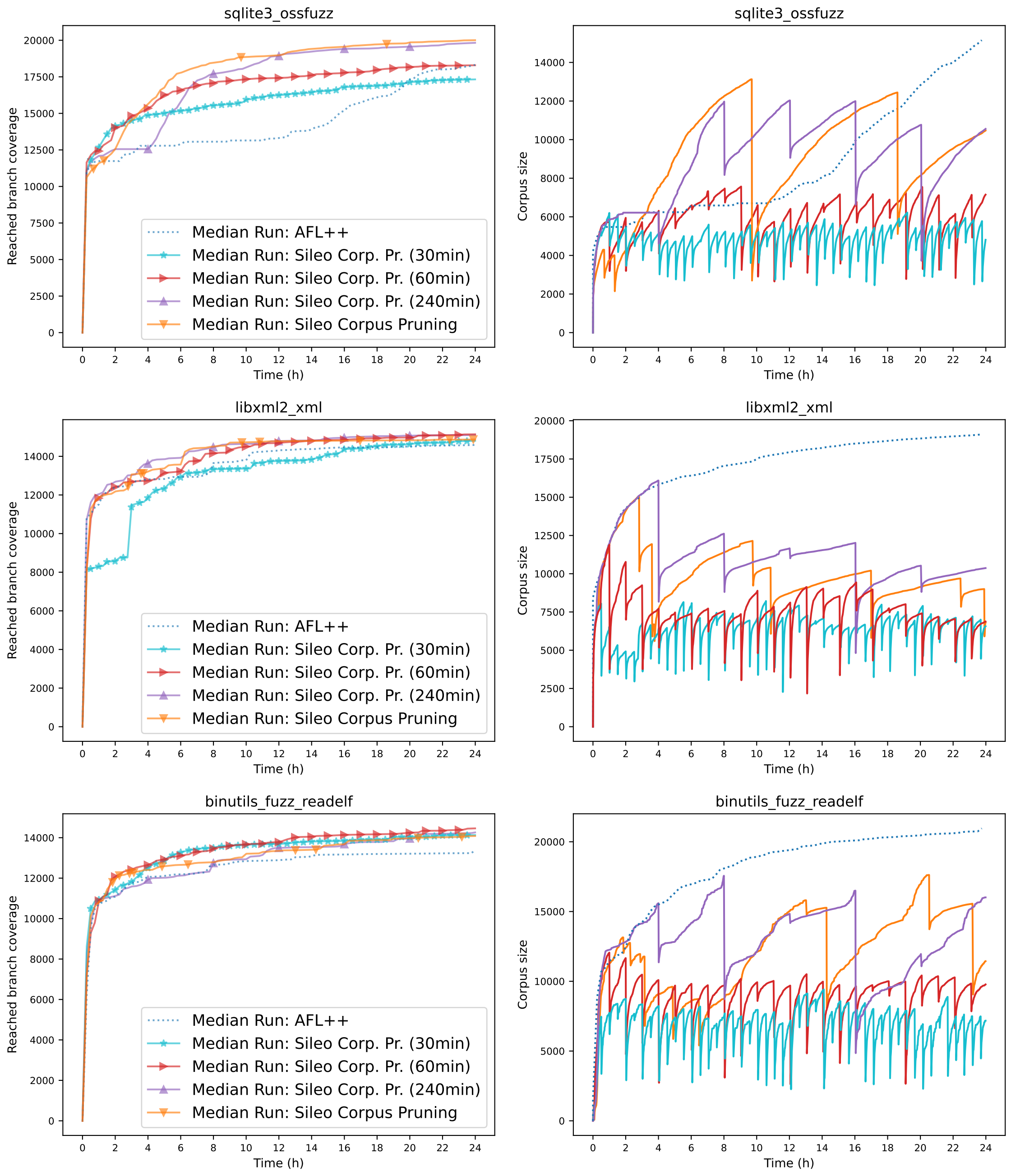


Fig. 8. **Left:** The coverage (in branches) from `sqlite3`, `libxml2`, and `readelf` with fixed restart times. **Right:** The corpus size of AFL++, `corpus pruning`, and `corpus pruning` with those fixed restart times to showcase the number of files of the corpus that are preserved by each restart. Displayed in each graph is the median run of ten repetitions, and for `Sileo` the restarts are marked.

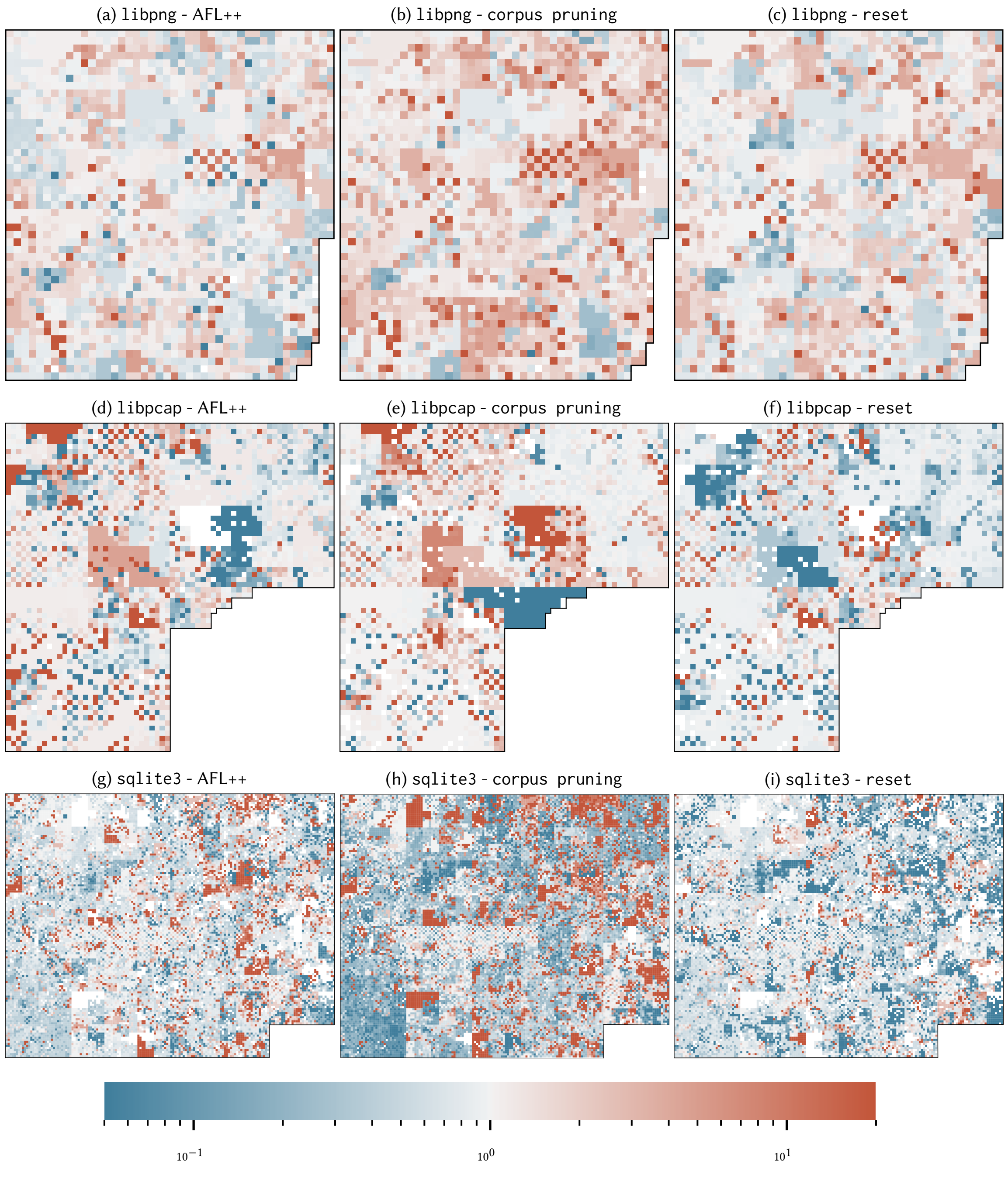


Fig. 9. Heatmaps showing the frequency of basic block hits during the median fuzzing run of AFL++, `corpus pruning`, and `reset` on the targets `libpng`, `libpcap`, and `sqlite3`. Each cell represents a basic block; blue blocks 🟦 indicate the median baseline run hit this basic block more frequently, and red 🟥 indicates the opposite.

while `reset` triggers only the most frequent 1, 500 edges more often than AFL++. On `sqlite3`, `corpus pruning` covers more edges than AFL++ and `reset`. Consequently, it is the only fuzzer triggering edges on the far right side of the CDF. Interestingly, the CDF shows that AFL++ outperforms our `Sileo` strategies on the most frequently hit 10, 000 edges, while `corpus pruning` hits its rarest 10,000 edges more frequently. We also observe this behavior when examining Table 3 and the heatmaps themselves (Figure 9), which illustrate that on `sqlite3`, AFL++ outperforms `corpus pruning` and `reset` by a considerable margin, triggering basic blocks more often but achieving less coverage compared to `corpus pruning`.

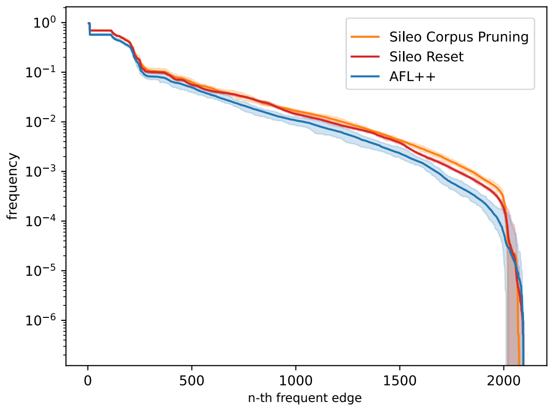
Interestingly, this behavior contradicts our observations from the targets `libpng` and `libpcap`. However, it can be explained when taking a closer look at the different heatmaps: On the two targets `libpng` and `libpcap`, all three fuzzers achieve roughly the same coverage, while the frequency of basic block hits is higher for `Sileo`. This happens because the target applications are smaller than `sqlite3`, and the coverage growth of all fuzzers flattens at an early stage of fuzzing. With every restart, mostly the same parts of the code are covered again, increasing the hit frequencies on these targets. Comparing this to `sqlite3`, which is significantly more complex than the other two targets, shows that the fuzzers do not flatline within the complete 24 hour run but continuously find new coverage. The dynamic nature of `sqlite3`, with continuous coverage growth, allows `corpus pruning` to trigger new and less frequent basic blocks even after the 24 hour mark. If `corpus pruning` restarts on this target, it breaks out of the coverage plateau and has the opportunity to (randomly) drive exploration into another part of the program. This behavior can be observed in the heatmaps in Figure 9, where `corpus pruning` is more focused on different "islands", while AFL++ distributes its executions over the rest of the program. Direct comparison shows that these islands of red blocks are the code parts that are not covered by AFL++ at all (indicated by a complete white rectangle blocks).

The experimental results provide valuable insights into the impact of input shadowing on different types of targets, effectively addressing **RQ1**. They demonstrate that a fuzzer can derive various benefits from a restart, depending on the complexity of the target, effectively mitigating the effects of input shadowing. The findings can be related to our example in Algorithm 1 (see Section 2.2): in case of a small target where the fuzzer quickly flatlines, costly executions are spent on mutating already known inputs to reach deeper code paths. However, some of these unexplored paths might be shadowed by previously found inputs, making progress unnecessarily difficult. Here, restarts can significantly increase

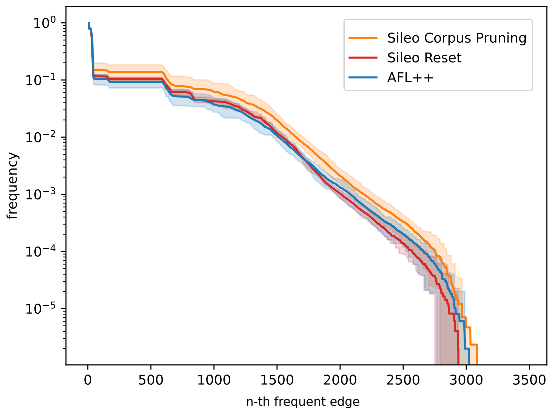
Table 3. Branch coverage from `libpng`, `libpcap`, and `sqlite3` of the median run from ten 24 hour runs and basic block hits that *more than twice* as often triggered by the baseline 🟦 or our restarted fuzzer 🟥. For `no-restarts` the *median* − 1 run is used. A gray background shows the best performing value in terms of median covered branches. Bold values indicate the best performance by means of favored basic blocks for the given strategy.

	Target	Restarts	Median #Branches	#Basic Blocks favoring 🟦 Baseline	#Basic Blocks favoring 🟥 Tweak
AFL++	`libpng`	–	2,004	–	–
	`libpcap`	–	2,970	–	–
	`sqlite3`	–	17,631	–	–
`no-restart`	`libpng`	0	2,004	229	**385**
	`libpcap`	0	2,950	545	545
	`sqlite3`	0	16,952	**5,383**	3,081
`corpus pruning`	`libpng`	21	2,014	122	**628**
	`libpcap`	13	2,938	305	**695**
	`sqlite3`	4	19,831	**10,341**	4420
`reset`	`libpng`	23	2,020	122	**522**
	`libpcap`	21	3,014	**687**	279
	`sqlite3`	16	15,642	**6,131**	1,556

(a) `libpng`



(b) `libpcap`



(c) `sqlite3`

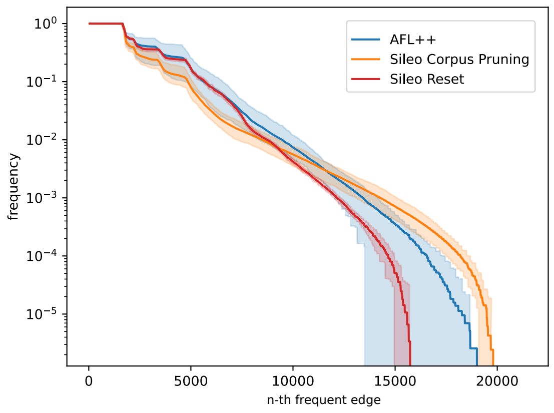


Fig. 10. Basic block frequency plot of `libpng`, `libpcap`, and `sqlite3` showing the strategies `corpus pruning`, `reset` and the baseline AFL++. Note that we assign a unique identifier to every edge and track how often each one is visited: x = 0 denotes the most frequent edge (one that is usually executed upon every execution e. g., the program's entry point). In contrast, the right-most value of x denotes the least frequently observed edge (often some edge never visited). On the y-axis, we then display how often the particular edge has been observed. A fuzzer is better if it covers more rarer edges (i. e., x is larger while y > 0) and visits rare edges more often.

the basic block hit frequencies, leading to a higher probability of encountering rare basic blocks more frequently and triggering that code more often. Conversely, if the fuzzer spends enough time on a small target application without restarting, it may eventually find an input that satisfies the given condition outlined in Algorithm 1. However, it likely needs more executions due to the lack of intermediate feedback. By restarting and discarding (parts of) its state, the fuzzer may accept a different input from the same equivalence class that may increase its probability of solving the given condition.

On complex targets, a similar yet distinct behavior can be observed. To exemplify, let us consider Algorithm 1 in the context of a complex target: A non-restarting fuzzer may trigger the first branch condition with a value that will never satisfy the second condition. With the lack of feedback and the complexity of the target, the fuzzer may shift its focus to other parts of the program or become stuck on that segment until both conditions are successfully solved. On this type of target, a restart has a similar effect as described previously, potentially triggering the shadowed code path and leading to more complex and deeper code exploration. However, on complex targets, a restart can also have a contrary effect: if the fuzzer is stuck in a deep path, a restart would allow the fuzzer to explore other parts of the code as well. This dual impact of restarts on complex targets emphasizes their flexibility in fostering exploration and uncovering hidden code paths.

Experiment 6: Bug-finding ability. The primary objective of a fuzzer is to identify vulnerabilities in software. Therefore, it is crucial to evaluate whether proposed changes to the fuzzing process have an impact—positive or negative—on the fuzzer's ability to discover bugs. As demonstrated by our coverage experiments, `Sileo` has proven its capability to achieve more coverage. However, it is unknown if this translates to a better bug-finding capability. In scenarios where the coverage of `Sileo` matches the one of the baseline, `Sileo` still exhibits the ability to trigger blocks more frequently, theoretically enhancing its chances of discovering more bugs. We conduct eight bug-finding experiments using FuzzBench, where a small amount of bugs is injected in specific targets, providing us with a ground truth. We summarize the results in Table 4. Since none of the fuzzers was able to trigger a bug on three out of eight targets (`arrow_parquet`, `file_magic`, and `mbedtls`), we omit these targets subsequently. The full report covering all targets can be found in our artifact. We analyzed the final corpora and the *crashes* directories of AFL++, `corpus pruning`, and `ensemble` to figure out how often the fuzzers were able to trigger crashes during the ten runs, and we depict these results in the table column *Runs w/ Bugs*. The results indicate that the `Sileo` strategies are able to trigger bugs in more runs and, according to FuzzBench, also found more bugs on all targets compared to our baseline AFL++. Notably, `corpus pruning` triggers bugs in seven out of ten runs for `bloatyfuzz` and in six out of ten runs for `aspell`, compared to only one run for AFL++ in each case. Additionally, `corpus pruning` triggered bugs in two out of ten runs for `libxml2`, where AFL++ and `ensemble` failed. Overall, both `Sileo` strategies were able to trigger bugs in more runs, which coincides with our results and our observations regarding input shadowing. These results allow us to positively answer **RQ3**, indicating that fuzzing restarts can help to discover more bugs in software.

5.2 Overhead

Restarting the fuzzing process introduces various types of overheads due to how we restart the underlying fuzzer. We measure disk usage, total fuzzing time, and total executions after 24 hours and plot this data in Figure 11. In addition to assessing three targets from our coverage evaluation, we also evaluated the overhead of our corpus minimization experiment. The AFL++ *fuzzer_stats* file provides measurements for executions (*execs_done*) and fuzzing runtime (*run_time*), while we determined disk usage over all queue files for AFL++ and all queue files plus new seed files per run for `Sileo`.

Disk Usage. In terms of disk usage, `Sileo` exhibits the most overhead compared to AFL++. Particularly, the disk usage of `input shuffle` reaches extremely high values, notably on `bloatyfuzz`, where AFL++ uses only 11MB, contrasting `Sileo`'s 36,870MB (3352 times the disk usage of AFL++). While lower, `corpus pruning` and `reset` still feature a disk usage multiple times that of AFL++. The primary issue lies in preserving the corpus from previous runs, where we store the queues of each restarted run.

With each restart, the fuzzer finds similar inputs that produce the same coverage as in previous runs, offering no further advantage but costing disk space (as we store the same files multiple times). This behavior is most pronounced in `input shuffle`, where the complete generated corpus is used and stored for the next run. Consequently, two identical

Table 4. Bugs found in 24 hour runs on five FuzzBench targets. We track the number of total unique bugs found, in how many runs one or more bugs were located, and statistics on the minimum, median, and maximum number of bugs per run. We grayed out the zeros to allow for better readability.

	Target	Median Coverage	Bugs				
			Unique	In Runs	Min	Median	Max
AFL++	bloatyfuzz	4569.0	1	1 / 10	0	0	1
	harfbuzz	10029.5	2	10 / 10	1	1	1
	libxml2	18572.0	0	0 / 10	0	0	0
	aspell	3192.0	2	1 / 10	0	1	2
	grok	5958.5	1	10 / 10	1	1	1
	Total		6	22 / 50	2	3	5
corpus pruning	bloatyfuzz	5560.0	1	7 / 10	0	1	1
	harfbuzz	10072.0	5	10 / 10	1	1	1
	libxml2	18793.0	1	2 / 10	0	0	1
	aspell	3437.0	2	6 / 10	1	2	2
	grok	6013.0	1	10 / 10	1	1	1
	Total		10	35 / 50	3	5	6
ensemble	bloatyfuzz	5520.5	1	6 / 10	0	1	1
	harfbuzz	10069.0	4	10 / 10	1	1	1
	libxml2	18788.5	0	0 / 10	0	0	0
	aspell	3215.5	2	4 / 10	0	2	2
	grok	6031.5	1	10 / 10	1	1	1
	Total		7	30 / 50	2	5	5

directories containing the same files are stored (the fuzzer queue and the new seed directory) for every run. Storing both directories is necessary because `input shuffle` renames all input files and adds a random prefix to each file, introducing an additional level of randomness to the seed processing of AFL++. An alternative approach to improve the `input shuffle` strategy in the future could be to use the last *queue* directory as the new seed directory without copying and renaming all inputs. This would reduce disk usage without affecting runtime, executions, or overall performance. When comparing `corpus pruning` and `input shuffle` with their respective `afl-cmin` versions, the data reveals that for `input shuffle`, `afl-cmin` reduces disk usage to approximately one-third of the run that does not utilize `afl-cmin`. Interestingly, for `corpus pruning`, the opposite trend is observed: disk usage increases slightly. We attribute this to the necessity for the fuzzer to rebuild the corpus from the remaining seeds after minimization without being able to explore new program behavior.

Total Fuzzing Time. When a restart occurs, the underlying fuzzer—in our case AFL++—is stopped, and our corpus retention strategies are applied to the generated corpus. The remaining files, which serve as new seed files for the next fuzzing run, are copied to the new seed directory. This process takes some time, and the processing of these seeds during the so-called *startup calibration phase* of AFL++ also reduces the overall fuzzing time. The reduction depends on the number of leftover files, ranging from a few seconds to almost an hour. In a 24 hour run with multiple restarts, this shortens the total fuzzing time. As depicted in Figure 11, the total fuzzing time is reduced by 1.5 hours for `input shuffle`, whereas the reduction is negligible for our other strategies.

Interestingly, `corpus pruning` has an overhead of 0.1% in terms of runtime compared to `reset` on `sqlite3`. This is due to the higher number of restarts for `reset` and the vast number of initial seed files that come with this specific target. The abundance of initial seeds, combined with the high number of restarts, results in slightly less fuzzing time for `reset`. Despite `corpus pruning` retaining both the generated corpus and the initial seeds, it has fewer restarts. Comparing `corpus pruning` and `input shuffle` with their respective `afl-cmin` versions shows that the reduction of inputs positively impacts `input shuffle`. The reduced number of inputs speeds up the startup calibration phase, compensating for the time `afl-cmin` takes for the corpus minimization itself. This is in contrast to `corpus pruning`, where the startup calibration phase reduces the overall fuzzing time by 0.1%, which is negligible here, but the corpus minimization adds an additional overhead of 0.2%. Since most of the time is lost during the startup calibration phase, we tested turning off this feature using the respective environment variable provided by AFL++. We found that this had a negative impact on the fuzzer in terms of coverage, so we refrained from using this setting for our evaluation.

Total Executions. Examining the total executions reveals that `Sileo` achieves significantly more executions than AFL++. We speculate that this occurs when the fuzzer, in our case AFL++, encounters a large number of initial seed files. This abundance of files leads to a spike in *execs per second* after restarting the fuzzing process because the startup calibration phase prioritizes testing the fastest seeds first. This effect is particularly pronounced on the target `sqlite3` with `input shuffle`, where the executions per second increase significantly due to the generation of a large corpus. However, despite the high number of executions, `input shuffle` does not gain any advantage over `corpus pruning` in terms of coverage. Comparing the executions of `corpus pruning` and `input shuffle` when using corpus minimization reveals a significant drop in the total number of executions for both. Upon investigating the corpus minimization statistics, we found that `afl-cmin` maintains the corpus size within the range of 2000 to 6000 corpus files for both strategies. Without `afl-cmin`, the number of corpus files reused as seeds can be much higher.

6 DISCUSSION

In the following, we discuss potential threats to validity, address the differences between code coverage and block frequency, and outline an optimized version of our `input shuffle` strategy.

Threats to Validity. It is essential to ensure the validity of the conclusions drawn from empirical experiments. Our research emphasizes three key dimensions that are particularly relevant to this goal. We outline our assumptions and describe our steps to ensure that our experiments maintain their validity.

External validity. To test our approach, we select different types of targets from Google's test suite FuzzBench for our evaluation. We used FuzzBench because it is a widely used benchmark framework for fuzzing. Furthermore, we also performed one of our coverage experiments (`objdump`) outside of FuzzBench to demonstrate that our results are reliable and not biased by using FuzzBench.

Internal validity. Due to the non-deterministic nature of the fuzzing process, we repeated our experiments ten times to achieve statistical significance and calculated several statistical metrics to analyze our experimental results. Furthermore, we used the well-established and proven fuzzing test suite FuzzBench to orchestrate and evaluate our experiments.

Construct validity. Finally, as a third threat to validity, we ensure that our evaluation measures what it is supposed to measure. To avoid discrepancies between the baseline and our tested strategies, we ensure they all rely on the same fuzzer configuration and the same fuzzer version of AFL++.

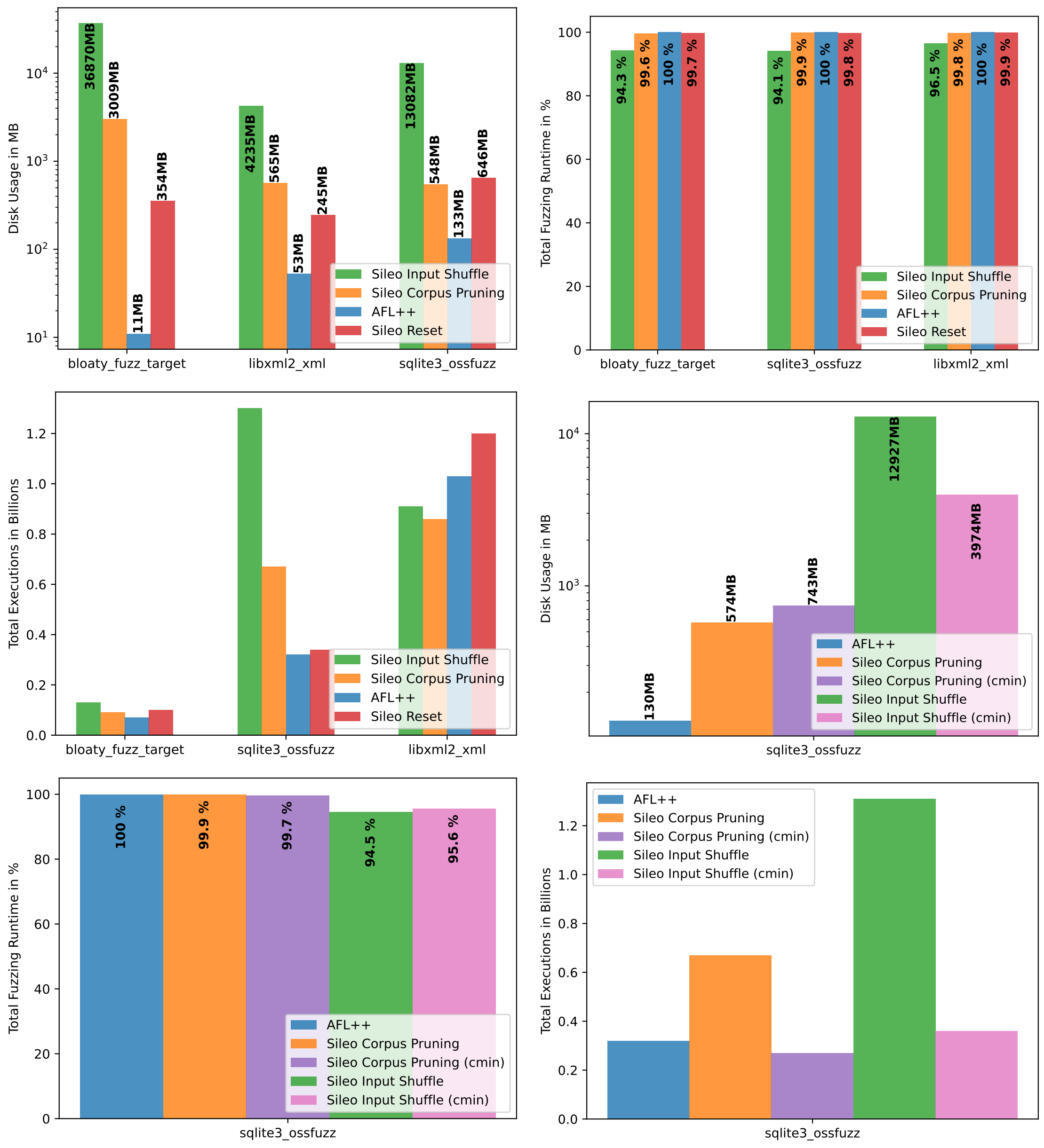


Fig. 11. Disk usage, total executions, and total fuzzer runtime of different targets for the median run.

Code Coverage vs. Block Frequency. In scientific literature and practice, code coverage is one of the standard metrics used to compare the performance of fuzzers. If a fuzzer achieves a high code coverage, it means that it can potentially trigger more bugs, as the fuzzer must reach buggy code in the first place. This paper introduces *block frequency* as a novel metric to augment, rather than replace, code coverage. Measuring *block frequency* can help understand which basic blocks the fuzzer has focused on the most. Visiting a basic block more often may be beneficial if the state between visits differs, as bugs may not occur for a specific state but can occur for another.

Input Shuffle with Reduced Overhead As discussed in Section 5.2, using `Sileo`'s corpus retention strategies introduces an overhead in terms of increased disk space usage. This is particularly pronounced when employing the `input shuffle` strategy, which significantly raises disk usage compared to all other strategies and AFL++. The design of these strategies involves copying the queue directory into a new seed directory, shuffling all files, and using this directory as the new seed directory for the underlying fuzzer. This copying is necessary to avoid modifying the *queue* directory of the run, such as when deleting files using `corpus pruning`. However, `input shuffle` differs from our other retention strategies in that nothing from the actual corpus is deleted; the entire corpus is used for the next run, and thus, no modification of the actual queue directory occurs, aside from shuffling all files. Given these considerations, we propose an optimized version of the `input shuffle` strategy: after a restart, this strategy sets the new seed directory to the old queue directory, thereby skipping the copy and shuffle process and minimizing the associated overhead. This strategy requires that the AFL++ environment variable *AFL_SHUFFLE_QUEUE* is set. We have tested this optimized strategy, and our initial results align with our expectations: the coverage performance is slightly higher compared to the one of `input shuffle`, and the strategy's overhead is reduced. We recommend using this strategy instead of `input shuffle` and providing the implementation alongside our artifact.

7 RELATED WORK

Our work is closely related to several previous works, and we now place our work in the context of the literature.

Resetting State. Resetting the state or starting over with a clean environment has been observed to have beneficial effects in various areas. For example, Zaidi et al. [69] discuss how reinitializing a neural network can significantly improve training results. They note that this phenomenon is surprisingly understudied and underused. In the field of program synthesis, Koenig et al. [33] recently showed that restarting stochastic synthesis tasks can boost the speed of synthesis by an order of magnitude. Similar to fuzzing, their initial observation is that synthesis tasks advance through a series of plateaus, which tend to be heavy-tailed and, thus, can get stuck without making further progress. Even in biology, clean "restarts" can be beneficial [35].

Also, in the research area of genetic algorithms and evolution strategies, (randomly) resetting the state to overcome local minima has a long history [14, 21, 23]. Ghannadian [23] presents a genetic algorithm that replaces traditional mutation operation with a random restart strategy. This implies that instead of mutating individuals in the population, the algorithm periodically restarts the search from a random point. This modification aims at improving exploration in the search space and avoiding premature convergence to local optima. However, Fukunaga [21] shows that the approach of scheduled restarts can be competitive over dynamic restart strategies, while pointing out that a combination of dynamic and scheduled restarts might be an interesting direction. This behavior can also be observed in our results, where we compared statically scheduled restarts (`corpus pruning`- 30min, 60min, 240min) with the dynamic restart approach of the same strategy. However, our results indicate that the number of statically scheduled restarts leading to high success depends on the individual target, making the dynamic approach more reliable in practice over all targets.

Coverage Plateaus. Similar to our approach, previous work has identified coverage plateaus as an excellent point in time to help the fuzzer. According to Lemieux et al. [36], the phenomenon known as *coverage stall* or *coverage plateau* occurs when a search algorithm, after undergoing several mutation iterations, fails to exhibit any further improvements in code coverage. Despite generating multiple mutated test cases, none of them succeed in covering any previously unexplored code within the program under test. Hence, the authors use the term "coverage stall" to describe this state. While `Sileo` opts to restart the fuzzer in such a case, Lemieux et al. [36] propose to use Large Language Models (LLMs) tailored towards code generation, such as `Codex`, to "unstuck" the fuzzer. Outside of the fuzzing community, other works show that restarts can be used to escape plateaus in boolean satisfiability problems [27].

While coverage plateaus worked well for our prototype, other indicators may exist. In particular, Liyanage et al. [38] recently discussed the extrapolation of coverage rates. While their goal was to find a stopping criteria for fuzzing, which may run indefinitely otherwise, we can repurpose any such approach and restart a fuzzer once the estimated coverage rate falls below some threshold.

Fuzzing. Fuzzing has a long and successful history, starting with Miller's initial work [44]. One particularly important hallmarks has been the introduction of coverage feedback, popularized by AFL [70], which spurred a vast body of subsequent research [8, 9, 49]. These works then focused on even more heavyweight feedback techniques such as taint tracking [12, 65] or symbolic execution [10, 50, 68], improving seed scheduling [9, 59], or proposing entirely new approaches [5, 11, 47, 53, 58, 63].

The success of fuzzing has not only lead to industry adoption [43] but has seen it ported to many other fields and types of applications than only Linux userspace binaries. In particular, work has been done to test kernels [24, 30, 31, 56, 62], network applications [4, 39, 45, 48, 55], JavaScript engines or other browser components [6, 25, 28, 60, 64, 67, 72], and firmware [1, 17, 51, 52, 57, 71]. With the rise of neural networks and large language models, new fuzzing approaches leveraging these techniques have been developed [15, 36, 41, 46, 58]. Recently, concurrent work by Wu et al. [66] uses a similar exemplary function for motivating their fuzzer design as we do in Algorithm 1. While we restart the fuzzer to mitigate input shadowing, their approach envisions the generation of so-called phantom programs, which reduce the nested conditional statements, so that they can be solved independently by the fuzzer, enabling it to make progress faster. Thus, their technique is closer to approaches such as laf-intel [34].

8 CONCLUSION

In this work, we present a starting point for mitigating the negative impact of input shadowing on fuzzing campaigns. We found that the novelty search used by fuzzing algorithms limits its practically reachable input space. To mitigate this downside of the otherwise excellent novelty search, we propose a fuzzing campaign scheduler called `Sileo` that restarts fuzzers adaptively to diversify their attention. Our results indicate that `Sileo` effectively distributes the hit block frequencies and better spreads the fuzzer's attention across different basic blocks. Restarting the fuzzing process proves beneficial in overcoming input shadowing. Inputs previously deemed non-interesting due to lack of novelty in terms of new coverage are given a renewed chance for exploration. The dynamic interaction between the bitmap coverage tracking, the preserved corpus, and the restart heuristic underscores `Sileo`'s adaptive approach, ensuring a more comprehensive and nuanced exploration of the program's input space.

ACKNOWLEDGMENTS

This work was funded by the European Research Council (ERC) under the consolidator grant RS^3 (101045669) and the German Federal Ministry of Education and Research (BMBF) under the grant KMU-Fuzz (16KIS1898).

REFERENCES

[1] Ioannis Angelakopoulos, Gianluca Stringhini, and Manuel Egele. FirmSolo: Enabling Dynamic Analysis of Binary Linux-based IoT Kernel Modules. In *USENIX Security Symposium*, 2023.

[2] Andrea Arcuri and Lionel Briand. A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering. In *International Conference on Software Engineering (ICSE)*, 2011.

[3] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[4] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful Greybox Fuzzing. In *USENIX Security Symposium*, 2022.

[5] Nils Bars, Moritz Schloegel, Tobias Scharnowski, Nico Schiller, and Thorsten Holz. Fuzztruction: Using Fault Injection-based Fuzzing to Leverage Implicit Domain Knowledge. In *USENIX Security Symposium*, 2023.

[6] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. JIT-Picking: Differential Fuzzing of JavaScript Engines. In *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[7] Tim Blazytko, Cornelius Aschermann, Moritz Schloegel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing Structure while Fuzzing. In *USENIX Security Symposium*, 2019.

[8] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2329–2344, 2017.

[9] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[10] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[11] Ju Chen, Jinghan Wang, Chengyu Song, and Heng Yin. JIGSAW: Efficient and Scalable Path Constraints Fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[12] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[13] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *USENIX Security Symposium*, 2019.

[14] Giuseppe Cuccu, Faustino Gomez, and Tobias Glasmachers. Novelty-based restarts for evolution strategies. In *2011 IEEE Congress of Evolutionary Computation (CEC)*, pages 158–163. IEEE, 2011.

[15] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, pages 423–435, 2023.

[16] Stephane Doncieux, Alban Laflaquière, and Alexandre Coninx. Novelty Search: A Theoretical Perspective. In *Genetic and Evolutionary Computation Conference (GECCO)*, 2019.

[17] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. Snipuzz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[18] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. The Use of Likely Invariants as Feedback for Fuzzers. In *USENIX Security Symposium*, 2021.

[19] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[20] Yu-Fu Fu, Jaehyuk Lee, and Taesoo Kim. autofz: Automated Fuzzer Composition at Runtime. In *USENIX Security Symposium*, 2023.

[21] Alex S Fukunaga. Restart scheduling for genetic algorithms. In *International Conference on Parallel Problem Solving from Nature*, pages 357–366. Springer, 1998.

[22] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: Path Sensitive Fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[23] Farzad Ghannadian, Cecil Alford, and Ron Shonkwiler. Application of random restart to genetic algorithms. *Information Sciences*, 95(1-2):81–102, 1996.

[24] google. syzkaller - kernel fuzzer. https://github.com/google/syzkaller. Accessed: June 12, 2024.

[25] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. Fuzzilli: Fuzzing for JavaScript JIT Compiler Vulnerabilities. In *Symposium on Network and Distributed System Security (NDSS)*, 2023.

[26] Emre Güler, Philipp Görz, Elia Geretto, Andrea Jemmett, Sebastian Österlund, Herbert Bos, Cristiano Giuffrida, and Thorsten Holz. Cupid: Automatic Fuzzer Selection for Collaborative Fuzzing. In *Annual Computer Security Applications Conference (ACSAC)*, 2020.

[27] Steven Hampson and Dennis Kibler. Plateaus and plateau search in boolean satisfiability problems: When to give up searching and start again. In *Workshop Notes: 2nd DIMACS Challenge*. Citeseer, 1993.

[28] Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, and Wei Huo. SoFi: Reflection-Augmented Fuzzing for JavaScript Engines. In *ACM Conference on Computer and Communications Security (CCS)*, 2021.

[29] Adrian Herrera, Mathias Payer, and Antony L Hosking. DatAFLow: Toward a Data-Flow-Guided Fuzzer. *ACM Transactions on Software Engineering and Methodology*, 2022.

[30] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding Kernel Race Bugs through Fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[31] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. HFL: Hybrid Fuzzing on the Linux Kernel. In *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[32] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[33] Jason R. Koenig, Oded Padon, and Alex Aiken. Adaptive Restarts for Stochastic Synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2021.

[34] lafintel. laf-intel - Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com.

[35] Joel Lehman and Risto Miikkulainen. Extinction Events can Accelerate Evolution. *PloS one*, 10(8):e0132886, 2015.

[36] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *International Conference on Software Engineering (ICSE)*, 2023.

[37] Caroline Lemieux and Koushik Sen. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018.

[38] Danushka Liyanage, Seongmin Lee, Chakkrit Tantithamthavorn, and Marcel Böhme. Extrapolating Coverage Rate in Greybox Fuzzing. In *International Conference on Software Engineering (ICSE)*, 2024.

[39] Zhengxiong Luo, Junze Yu, Feilong Zuo, Jianzhong Liu, Yu Jiang, Ting Chen, Abhik Roychoudhury, and Jiaguang Sun. Bleem: Packet Sequence Oriented Fuzzing for Protocol Implementations. In *USENIX Security Symposium*, 2023.

[40] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized Mutation Scheduling for Fuzzers. In *USENIX Security Symposium*, 2019.

[41] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024.

[42] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[43] Mike Aizatsky, Kostya Serebryany (Software Engineers, Dynamic Tools); Oliver Chang, Abhishek Arya (Security Engineers, Google Chrome); and Meredith Whittaker (Open Research Lead). Announcing OSS-Fuzz: Continuous fuzzing for open source software. https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html, 2016.

[44] Barton P Miller, David Koski, Cjin Pheow Lee, Vivekandanda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1995.

[45] Roberto Natella. StateAFL: Greybox Fuzzing for Stateful Network Servers. *Empirical Software Engineering*, 27(7):191, 2022.

[46] Yaroslav Oliinyk, Michael Scott, Ryan Tsang, Chongzhou Fang, Houman Homayoun, et al. Fuzzing busybox: Leveraging llm and crash reuse for embedded bug unearthing. *arXiv preprint arXiv:2403.03897*, 2024.

[47] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: Fuzzing by Program Transformation. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[48] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. AFLNet: A Greybox Fuzzer for Network Protocols. In *IEEE International Conference on Software Testing, Validation and Verification (ICST)*, 2020.

[49] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 47(9):1980–1997, 2019.

[50] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with SymCC: Don't interpret, compile! In *USENIX Security Symposium*, 2020.

[51] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing. In *USENIX Security Symposium*, 2022.

[52] Tobias Scharnowski, Simon Woerner, Felix Buchmann, Nils Bars, Moritz Schloegel, , and Thorsten Holz. Hoedur: Embedded Firmware Fuzzing using Multi-Stream Inputs. In *USENIX Security Symposium*, 2023.

[53] Nico Schiller, Merlin Chlosta, Moritz Schloegel, Nils Bars, Thorsten Eisenhofer, Tobias Scharnowski, Felix Domke, Lea Schönherr, and Thorsten Holz. Drone security and the mysterious case of dji's droneid. In *NDSS*, 2023.

[54] 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.

[55] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *USENIX Security Symposium*, 2021.

[56] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *USENIX Security Symposium*, 2017.

[57] Lukas Seidel, Dominik Maier, and Marius Muench. Forming Faster Firmware Fuzzers. In *USENIX Security Symposium*, 2023.

[58] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[59] Dongdong She, Abhishek Shah, and Suman Jana. Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[60] Suhwan Song, Jaewon Hur, Sunwoo Kim, Philip Rogers, and Byoungyoung Lee. R2Z2: Detecting Rendering Regressions in Web Browsers through Differential Fuzz Testing. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2022.

[61] András Vargha and Harold D Delaney. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[62] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael B. Abu-Ghazaleh. SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning. In *USENIX Security Symposium*, 2021.

[63] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2019.

[64] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware Greybox Fuzzing. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2019.

[65] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A Checksum-aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[66] Mingyuan Wu, Kunqiu Chen, Qi Luo, Jiahong Xiang, Ji Qi, Junjie Chen, Heming Cui, and Yuqun Zhang. Enhancing coverage-guided fuzzing via phantom program. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 1037–1049, 2023.

[67] Wen Xu, Soyeon Park, and Taesoo Kim. FREEDOM: Engineering a State-of-the-Art DOM Fuzzer. In *ACM Conference on Computer and Communications Security (CCS)*, 2020.

[68] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium*, 2018.

[69] Sheheryar Zaidi, Tudor Berariu, Hyunjik Kim, Jörg Bornschein, Claudia Clopath, Yee Whye Teh, and Razvan Pascanu. When Does Re-initialization Work? In *I Can't Believe It's Not Better Workshop at NeurIPS*, 2022.

[70] Michał Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/. Accessed: June 12, 2024.

[71] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *USENIX Security Symposium*, 2019.

[72] Chijin Zhou, Quan Zhang, Mingzhe Wang, Lihua Guo, Jie Liang, Zhe Liu, Mathias Payer, and Yu Jiang. Minerva: Browser API Fuzzing with Dynamic mod-ref Analysis. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022.