Novelty Not Found: Adaptive Fuzzer Restarts to Improve Input Space Coverage (Registered Report)

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

ABSTRACT

Feedback-driven greybox fuzzing is one of the cornerstones of modern bug detection techniques. Its flexibility, automated nature, and effectiveness render it an indispensable tool for making software more secure. A key feature that enables its impressive performance is coverage feedback, which guides the fuzzer to explore different parts of the program. The most prominent way to use this feedback is *novelty search*, in which the fuzzer generates new inputs and only keeps those that have exercised a new program edge. This is grounded in the assumption that novel coverage is a proxy for interestingness. Bolstered by its widespread success, it is easy to overlook its limitations. Particularly the phenomenon of *input shadowing*, i. e., situations in which an "interesting" input is discarded because it does not contribute novel coverage, needs to be considered. This phenomenon limits the explorable input space and risks missing bugs when shadowed inputs are more amenable to mutations that would trigger bugs.

In this work, we analyze input shadowing in more detail and find that multiple fuzzing runs of the same target exhibit a different basic block hit frequency despite overlapping code coverage. In other words, different fuzzing runs may find the same set of basic blocks but one might exercise specific basic blocks significantly more often than the other, and vice versa. To better distribute the block frequency, we propose restarting the fuzzer to reset the fuzzing state, diversifying the fuzzer's attention across basic blocks. Our preliminary evaluation of three FuzzBench targets finds that fuzzer restarts effectively distribute the basic block hit frequencies and boost the achieved coverage by up to 9.3%.

CCS CONCEPTS

• **Security and privacy** → Software security engineering.

FUZZING '23, July 17, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0247-1/23/07…$15.00
https://doi.org/10.1145/3605157.3605171

KEYWORDS

Fuzzing, Software Security

ACM Reference Format:
Nico Schiller, Xinyi Xu, Lukas Bernhard, Nils Bars, Moritz Schloegel, and Thorsten Holz. 2023. Novelty Not Found: Adaptive Fuzzer Restarts to Improve Input Space Coverage (Registered Report). In *Proceedings of the 2nd International Fuzzing Workshop (FUZZING '23), July 17, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3605157.3605171

1 INTRODUCTION

Fuzzing is a highly effective technique for uncovering bugs 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 [24], fuzzing has evolved into a highly specialized, well-developed technique that has undergone significant improvements in the past few years. Especially feedback-driven greybox fuzzing, as popularized by AFL [35], 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 possible. 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.

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 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. The second input, however, could be an essential precursor to reach an interesting program location, even when not exploring new coverage. 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. Yet, 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 [14], datAFLow [16], or InvsCov [11] implicitly attempt to *circumvent* this problem by using more fine-granular or different feedback mechanisms. They increase the likelihood of considering inputs novel according to their metric, which were previously shadowed. 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 farther gaps in the input space, as depicted in Figure 1. This can either be archived by dividing the gap into several smaller ones (e. g., via `cmplog` [2, 12]) or by improved mutations. Intuitively, we can use target-specific mutations to bridge farther gaps and, thus, increase the reachable input space [4, 21, 22]. 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 other 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 potentially jammed further fuzzer progress. While this appears counterintuitive, as information accumulated by costly executions is discarded, our preliminary results indicate that this technique is beneficial in diversifying the basic blocks tested and helping the fuzzer find new coverage.

In summary, we make the following key contributions:

- 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.

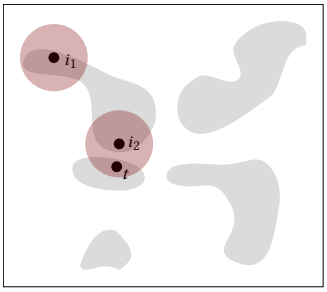

Figure 1: Input space of a target program. Grey areas mark areas leading to the same coverage, red circles represent the input space we can reach with our mutations given the respective input. As illustrated, i_1 and i_2 exercise the same coverage, however, our mutations can derive input t only from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

- We publish our code and artifacts at https://github.com/CISPA-SysSec/fuzzing-restarts.

2 MOTIVATION

Before presenting our design, we discuss 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 [10]. 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*, discussed in the next section.

2.2 Input Shadowing

We define input shadowing as a situation in which an input is discarded because it does not contribute unique coverage, i. e., it is not novel. Suppose we have an input i_1 that is novel, and another input i_2 achieves the same coverage; furthermore, i_2 is also an essential predecessor for some hypothetical inputs that will trigger some bug. We say that both inputs share a *novelty equivalence class* and that i_1 *shadows* i_2, since i_2 will be discarded due to lack of novelty. We emphasize that i_2 is not necessarily better or worse than the first input, i_1 – the difference is only at the byte level. This difference allows the fuzzer's mutations to create other inputs from i_2 than from i_1. Consider an example program with the input space 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 share the same coverage and i_1 is discovered first, we 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 empirically study this phenomenon for a fuzzing campaign of `libpng` by recording basic blocks hit during the fuzzing runs. More precisely, we run AFL++ multiple times for 24 hours using four different tweaks: As a baseline, we use a standard setup of

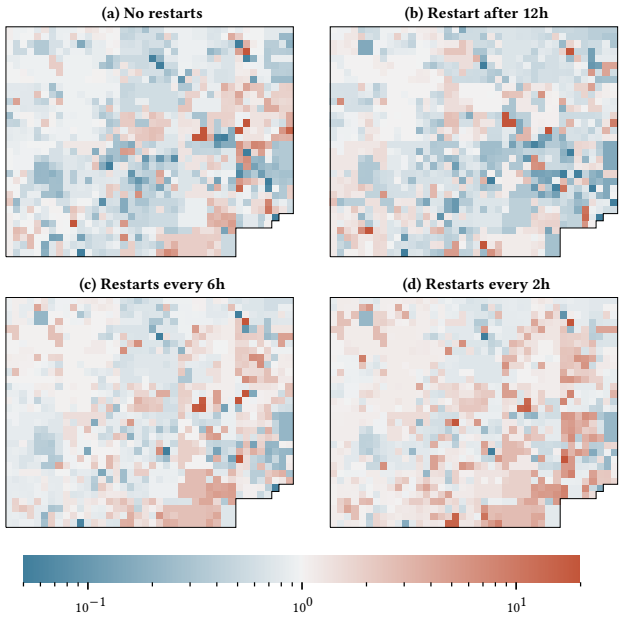

Figure 2: Heatmaps based on Hilbert curves showing the frequency of basic blocks hit during a fuzzing run relative to a baseline run (we use the median run from ten 24h repetitions for each). Each cell represents a basic block; blue basic blocks ■ indicate the median baseline run hit this basic block 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 blocks.

AFL++ without any restarts. Additionally, we use three further setups with restart frequencies of 12, 6, and 2 hours, resulting in 1, 3, and 11 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 and Figure 2. We sample every 10,000-th input during the runs and record the basic blocks it hits. Importantly, we do not work on the queue to avoid survivorship bias. As each input accepted into the corpus is novel, if we sample 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 using Hilbert curves in Figure 2, comparing the frequency of basic blocks hit against the baseline without any restarts. Each cell marks a distinct basic block; blue ■ indicates the block was hit more often by the baseline, while red ■ denotes the 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. Crucially, all represented fuzzing runs (baseline and restarted versions) exercised comparable coverage (see Table 1).

When running this experiment, one may intuitively assume that fuzzing runs with roughly equal coverage would exercise basic blocks at approximately the same frequency. However, when we

Table 1: Block coverage of the median run from ten 24h runs and blocks covered *more than twice* as often by the baseline ■ or our restarted fuzzer ■.

	Median	#basic blocks favoring ■ baseline	■ tweak
`baseline`	1,069	–	–
`no-restart`	1,068	267	113
`reset 12h`	1,139	306	77
`reset 6h`	1,167	124	167
`reset 2h`	**1,187**	75	227

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 a very good proxy for the observed program states.

The key insight to tackle the problem of input shadowing is that distinct fuzzing runs exhibit different block-hitting frequencies. Intuitively, this difference in basic blocks that the fuzzer focuses on implies that distinct inputs are generated and executed. Hence, the likelihood of different fuzzing runs picking all the same input from a novelty equivalence class, e. g., all fuzzing runs picking i_1 first and not i_2, is low in the general case. We empirically verify this hypothesis by comparing the corpora of two non-restarted fuzzing runs on the byte level. After using SHA-256 to hash all inputs of the respective corpora, we find that all inputs of two different runs are unique (except for the seed files). As the coverage of the runs is almost equal (cf. Table 1), this indicates the two runs selected different inputs from the novelty equivalence classes. On an abstract level, we can consider that running the fuzzer is a dice throw leading to some corpus. Throwing the dice again, i. e., running the fuzzer a second time, may yield the same coverage, albeit the corpus is likely to contain different inputs.

We observe that this ability to generate a different corpus without coverage loss mitigates the problem of input shadowing. As a result, re-running the fuzzer can be a worthwhile strategy to maximize the utilization of different inputs. We note this is already commonly done in fuzzing to mitigate the effects of randomness introduced by the fuzzing process. This observation of input shadowing also explains the common wisdom that a fuzzer may find a vulnerability in some runs but not others. Despite exercising similar coverage, it may find the inputs suited to trigger the bug in some runs; in others, it discards the potential input because the input is shadowed by another input from the same novelty equivalence class.

One problem with running the fuzzer multiple times is the associated cost: Running the fuzzer a second time doubles the cost, rendering the benefit of having potentially some different inputs

tested unclear. To avoid this cost, we could restart the fuzzer after half the runtime, say after 12 hours. This would presumably create a better frequency distribution of basic blocks hit, however, at the cost of potentially missing out on coverage. To investigate whether this works in practice, we restarted the fuzzer one time (after 12 hours), three times (after 6 hours each), and eleven times (after 2 hours each), resulting in Figures 2b, 2c, and 2d, respectively. As these heatmaps show, a higher number of restarts is indeed helpful to diversify the set of hit basic blocks. Interestingly, a single restart after 12 hours (Figure 2b) *worsens* the block frequency. On the other hand, 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 mitigate input shadowing and have a positive, yet minor impact on found coverage at the same time. We speculate that with a better, more adaptive strategy than restarting the fuzzer after a fixed time period, we can further increase the benefits.

2.3 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 it 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 application compartments, 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 behind 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.

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 demonstrate 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 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, which empirically works well in our experience.

Detection heuristics. Our scheduler features multiple strategies to decide 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 compartment of the target application.

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.

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.

In summary, our restart scheduler features three heuristics:

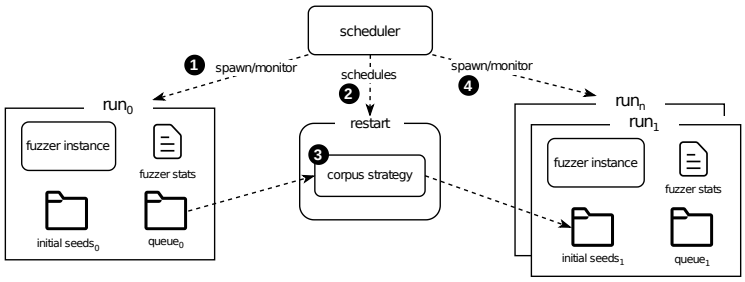

Figure 3: Overview of our restart scheduler

(1) Blind restarts with fixed or random reset countdown
(2) Coverage plateau-based resets
(3) Coverage growth rate-based resets 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) 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.

(iii) Timeback. The `timeback` 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.

(iv) 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 is 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.

(v) 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.

(vi) Ensemble. Similar to ensemble fuzzing, where multiple fuzzers are combined and scheduled adaptively [9, 15], we hypothesize that an `ensemble` strategy, which dynamically selects the most suitable strategy for the respective target under test, could improve the results. If all restart conditions are met, we use a predefined strategies priority and the accumulated performance of each used strategy from the previous runs to decide whether we should replace the current strategy.

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 1,027 lines of Python code and uses AFL++ in version *4.06a* 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 paths 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. Once the fuzzing campaign ends, the coverage of the fuzzer are calculated over all runs via `llvm-cov`.

For our preliminary experiments, we used FuzzBench [23]. 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 modify FuzzBench to only store 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 matrices over the generated fuzzer test cases.

5 PRELIMINARY EVALUATION

For our preliminary evaluation, we use our prototype `Sileo` and test it with different heuristics to decide when to restart the fuzzer and various corpus retention strategies to process the corpus. We first discuss the preliminary experiments we have already conducted before outlining the experiments we plan to run upon acceptance of this report in Section 5.3.

Experiment Setup We perform our evaluation on a server with an Intel Xeon Gold 6230R CPU with 52 cores at 2.10GHz and 196GB memory, running Ubuntu 22.04. We use FuzzBench to orchestrate the fuzzing campaign and run each strategy as an individual fuzzer. We ran all fuzzers for 24 hours and repeated each experiment ten times to account for inherent randomness in the fuzzing process, as recommended by Klees et al. [17]. The fuzzer we use in our experiments and as a baseline to compare our scheduler to is AFL++ in version 4.06a.

To evaluate the impact of restarting the fuzzer on the code coverage it exercises, we conducted a preliminary evaluation on three diverse programs from FuzzBench: `libpng`, `libpcap`, and `sqlite`. To test `Sileo`, we choose two basic strategies to showcase the impact (i) of clean restarts after the coverage stagnates without any parts of the corpus being preserved (`reset`) and (ii) of restarts that rely on the previous corpus (`corpus pruning`). Both strategies are restarted when a randomly chosen countdown (between 5min and 30min) elapses after detecting a coverage plateau without significant coverage growth. To identify slow coverage growth, we set the threshold to 2% of the current restart run coverage. If the restart countdown elapses and the average coverage growth drops below the threshold for some time (here, we empirically set this period to 5 minutes), we then issue a restart.

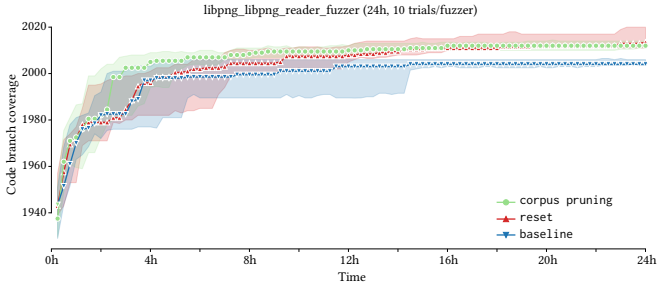

Figure 4: Coverage plot of `libpng` showing the strategies `corpus pruning` and `reset` compared to the baseline AFL++.

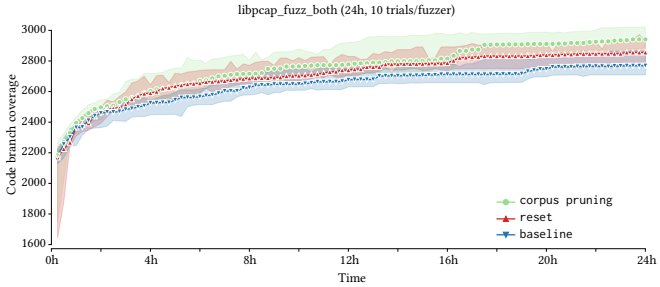

Figure 5: Coverage plot of `libpcap` showing the strategies `corpus pruning` and `reset` compared to the baseline AFL++.

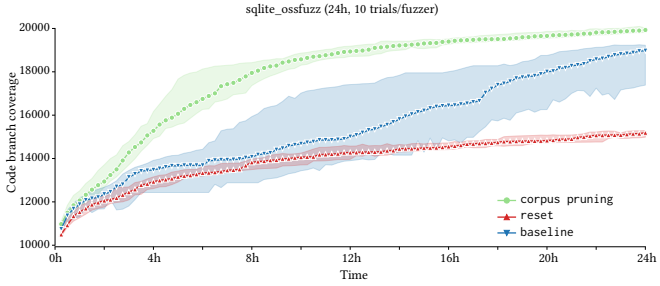

Figure 6: Coverage plot of `sqlite` showing the strategies `corpus pruning` and `reset` compared to the baseline AFL++.

5.1 Code Coverage

We plot the coverage in Figures 4, 5, and 6, respectively. Across all three targets, the results show that restarts via the corpus strategy `corpus pruning` lead to higher code coverage, outperforming both the baseline AFL++ and the `reset` strategy. On the other hand, the performance of the `reset` strategy seems to depend on the target: while it performs worse than the baseline on *sqlite*, it performs similarly on the other two targets. Compared to `corpus pruning`, `reset` performs worse on `sqlite` and `libpcap`, but has a slight advantage on `libpng`.

To confirm our results, we follow Arcuri's and Briand's recommendation [1] 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 [30]. We depict them in Table 2. For `corpus pruning`, all results regarding the baseline are statistically significant and the effect sizes are large, while for our second strategy, `reset`, only two cases are statistically significant, in one of which

Table 2: Statistical analysis of our code coverage experiment, with the hypothesis that tweak performs better than baseline. We use the median coverage 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) [30]. We use the two-sided non-parametric Mann-Whitney-U test and mark $p < 0.05$ in bold.

tweak	baseline	target	effect size	p-value
`corpus pruning`	baseline	`sqlite`	+L(0.99)	< **0.0001**
		`libpcap`	+L(0.88)	**0.0029**
		`libpng`	+L(0.98)	< **0.0001**
reset	baseline	`sqlite`	-L(0.10)	**0.0015**
		`libpcap`	+M(0.71)	0.1230
		`libpng`	+L(0.98)	< **0.0001**
`corpus pruning`	reset	`sqlite`	+L(1.00)	< **0.0001**
		`libpcap`	+M(0.71)	0.1230
		`libpng`	-S(0.43)	0.6842

Table 3: Number of restarts over ten runs (med. = median).

Target	Strategy	#restarts min	avg	max	med.
`sqlite`	`corpus pruning`	4	4.7	6	5
	`reset`	18	19.0	22	18
`libpng`	`corpus pruning`	18	18.9	20	19
	`reset`	12	13.1	14	13
`libpcap`	`corpus pruning`	11	12.7	15	13
	`reset`	17	19.4	21	20

the effect size is large but negative. When comparing our strategies, we find that the observed differences are statistically significant only for `sqlite`, where a large effect size is observed. Even though `reset` slightly outperforms `corpus pruning` on `libpng`, the difference is not statistically significant.

To introspect our techniques, we track the number of restarts performed by `corpus pruning` and `reset` throughout the evaluation in Table 3. Interestingly, neither strategy seems to have a definitive edge over the other, with the number of restarts being similar. `corpus pruning` features more restarts on `libpng`, `reset` on `sqlite` and `libpcap`. When relating this data to the code coverage, we find the number of restarts appears to have fewer for the strategy finding more coverage. For `libpng` and `libpcap`, this effect is barely visible as both strategies find very similar coverage. For `sqlite`, on the other hand, `corpus pruning` was restarted only five times, yet finds significantly more coverage than `reset`. This indicates that `corpus pruning` continuously found new coverage, while `reset` did not. This implies keeping state is beneficial on `sqlite`, suggesting that the fuzzer finds it more difficult to unlock new coverage on `sqlite` than on `libpng` or `libpcap`.

Our empirical evaluation shows that different restart strategies significantly impact the results. We find that `corpus pruning` outperforms `reset` for two targets and is superior to the baseline for all targets, indicating this is a promising strategy to increase code coverage.

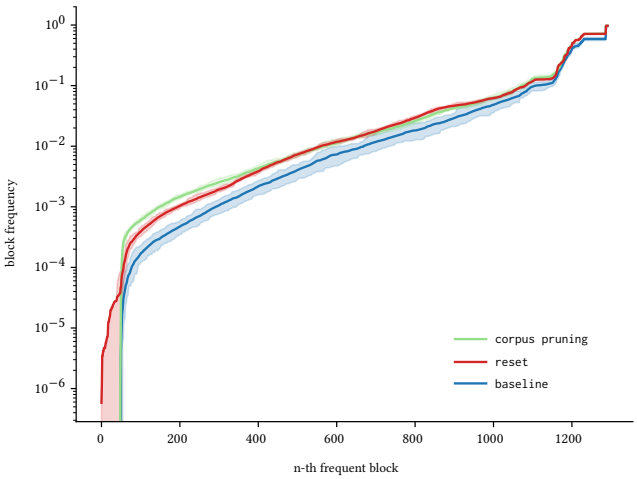

Figure 7: Basic block frequency plot of `libpng` showing the strategies `corpus pruning`, `reset` and the baseline AFL++. Lower values of n (x-axis) indicate rarely hit basic blocks, higher values of n represent basic blocks hit more frequently.

5.2 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., better spreading the fuzzer's attention to a diverse set of basic blocks. To this end, we use the inputs sampled during the fuzzing runs (similar to the experiment in Section 2). We do not use the corpus to avoid biasing our results. The plot in Figure 7 illustrates the distribution of basic block hits as a Cumulative Distribution Function (CDF). It displays the least frequent hit basic blocks on the left and the basic blocks hit most frequently on the right. For example, the program's entry point is the most commonly hit basic block and, thus, is positioned at the far right side of the plot. We then plot the frequency of all basic blocks being hit on the y-axis. As shown in Figure 7, both `reset` and `corpus pruning` trigger basic blocks more frequently than the baseline, particularly for less frequent basic blocks. This demonstrates that restarting the fuzzer and reusing parts of the corpus increases the probability of encountering rare basic blocks more frequently. Interestingly, roughly fifty basic blocks have neither been found by `corpus pruning` nor `baseline`, but only by the `reset` strategy.

Distributing the fuzzer's attention more evenly across basic blocks potentially enhances the coverage and bug-finding capabilities of the fuzzer. By enabling the fuzzer to trigger rare basic blocks more often, it can also explore new coverage areas associated with such edges. Due to the cost associated with sampling, we restrict our analysis of block frequency to `libpng`, but we will extend this experiment to all three targets for the full version of this work.

5.3 Planned Evaluation

We plan to extend our evaluation in several aspects to substantiate our hypotheses further and evaluate how restarting fuzzers impacts fuzzing runs. In particular, we want to add new targets, test all strategies introduced in Section 3, and test the bug-finding ability.

New targets and strategies. Our preliminary evaluation focuses on a subset of three targets from FuzzBench to limit computational resources spent. We plan to extend our evaluation to six diverse targets from the FuzzBench dataset. More precisely, we pick `freetype`, `lcms_cms`, and `libxml2`, as other targets exhibit few differences in achieved coverage between fuzzers in the latest FuzzBench reports [13]. Beyond FuzzBench, we intend to fuzz three commonly fuzzed real-world programs, e. g., `7zip`, `pdftotext`, and `objdump`. In total, we want to evaluate nine targets.

We intend to test the combination of all six strategies and heuristics presented in Section 3 for all nine targets. We have previously motivated why certain strategies could be worthwhile and plan to empirically evaluate whether our assumptions hold using the complete set of targets. Since we observed that the performances of the corpus strategies depend on the target, a more advanced ensemble approach that chooses the best strategy to clean up the corpus after restarting may be beneficial.

Bug-finding ability and long-term runs. As the primary goal of fuzzing is to uncover bugs, our future evaluation aims to analyze the potential enhancement in bug-finding capabilities through the use of fuzzing restarts. Considering the phenomenon of input shadowing, we hope that restarts can facilitate the exploration of unique code paths and increase the frequency of triggering vulnerable code. Additionally, we intend to assess the impact of restarts on complex targets during prolonged fuzzing campaigns.

6 DISCUSSION

In the following, we discuss potential threats to validity and address the differences between code coverage and block frequency.

Threats to Validity. Ensuring the validity of conclusions drawn from empirical experiments is essential. 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 preliminary evaluation. We used FuzzBench because it is a widely used benchmark framework for fuzzing. For our future evaluation, we plan to extend our evaluation to more FuzzBench targets and evaluate our approach on widely used targets such as `objdump`, `7zip`, and `pdftotext`. Furthermore, we plan to test our approach on other types of fuzzing targets, such as Javascript JIT engines, to study the potential benefits of our approach.

Internal validity. Due to the non-determinism nature of the fuzzing process, we repeated our experiments ten times to achieve statistical significance and calculated several statistical metrics to quantify 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 that they all rely on the same fuzzer configuration and the same fuzzer version of AFL++.

Code coverage vs. block frequency. In both the 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.

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. [34] discusses 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. [18] 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 [19].

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. [20], 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. [20] propose to use Large Language Models (LLMs) tailored towards code generation, such as `Codex`, to "unstuck" the fuzzer.

Fuzzing. Fuzzing has a long and successful history, starting with Miller's initial work [24]. Important hallmarks include the introduction of coverage feedback, popularized by AFL [35], which spurred a vast body of subsequent research. Directions covered include even more heavyweight feedback techniques such as taint tracking [8, 32] or symbolic execution [6, 26, 33], improved seed scheduling [5, 29], or entirely new approaches [3, 7, 25, 27, 28, 31].

8 CONCLUSION

In this work, we present a starting point for mitigating the 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 effect, we propose a fuzzing campaign scheduler called `Sileo` that restarts fuzzers adaptively to diversify their attention. Our preliminary results on a subset of FuzzBench indicate that `Sileo` effectively distributes the hit block frequencies and better spreads the fuzzer's attention across different basic blocks.

REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

[13] Fuzzbench. 2023. FuzzBench: 2023-04-27-main report. https://www.fuzzbench.com/reports/experimental/2023-04-27-main/index.html.

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

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

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

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

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

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

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

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

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

[23] Jonathan Metzman, László Szekeres, Laurent Maurice Romain Simon, Read Trevelin Sprabery, and Abhishek Arya. 2021. 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*.

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

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

[26] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic Execution with SymCC: Don't Interpret, Compile!. In *USENIX Security Symposium*.

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

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

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

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

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

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

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

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

[35] Michał Zalewski. [n. d.]. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/. Accessed: July 18, 2023.

Received 2023-05-15; accepted 2023-06-12