Accelerating Fuzzing through Prefix-Guided Execution

SHAOHUA LI, ETH Zurich, Switzerland
ZHENDONG SU, ETH Zurich, Switzerland

Coverage-guided fuzzing is one of the most effective approaches for discovering software defects and vulnerabilities. It executes all mutated tests from seed inputs to expose coverage-increasing tests. However, executing all mutated tests incurs significant performance penalties—most of the mutated tests are discarded because they do not increase code coverage. Thus, determining if a test increases code coverage *without actually executing it* is beneficial, but a paradoxical challenge. In this paper, we introduce the notion of *prefix-guided execution* (PGE) to tackle this challenge. PGE leverages two key observations: (1) Only a tiny fraction of the mutated tests increase coverage, thus requiring full execution; and (2) whether a test increases coverage may be accurately inferred from its partial execution. PGE monitors the execution of a test and applies early termination when the execution prefix indicates that the test is unlikely to increase coverage.

To demonstrate the potential of PGE, we implement a prototype on top of AFL++, which we call AFL++-PGE. We evaluate AFL++-PGE on MAGMA, a ground-truth benchmark set that consists of 21 programs from nine popular real-world projects. Our results show that, after 48 hours of fuzzing, AFL++-PGE finds more bugs, discovers bugs faster, and achieves higher coverage. Prefix-guided execution is general and can benefit the AFL-based family of fuzzers.

CCS Concepts: • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: fuzzing, code coverage, software testing

ACM Reference Format:
Shaohua Li and Zhendong Su. 2023. Accelerating Fuzzing through Prefix-Guided Execution. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 75 (April 2023), 27 pages. https://doi.org/10.1145/3586027

1 INTRODUCTION

Fuzzing has rapidly become one of the most popular automated testing techniques due to its simplicity and effectiveness in finding software defects and vulnerabilities [Aizatsky et al. 2016; Manès et al. 2019]. State-of-the-art fuzzing efforts center on coverage-guided fuzzing (CGF), *e.g.*, American Fuzzy Lop (AFL) [Zalewski 2014]. CGF works by generating a large number of tests from mutating seed inputs. All tests are executed on a target binary, and those coverage-increasing tests will be added to the seed pool for further exploration. Many previous efforts have shown that only a tiny fraction of the generated tests increase code coverage, *e.g.*, fewer than 1 in 10,000 [Nagy and Hicks 2019]. Thus, the current practice of blindly executing all generated tests is wasteful.

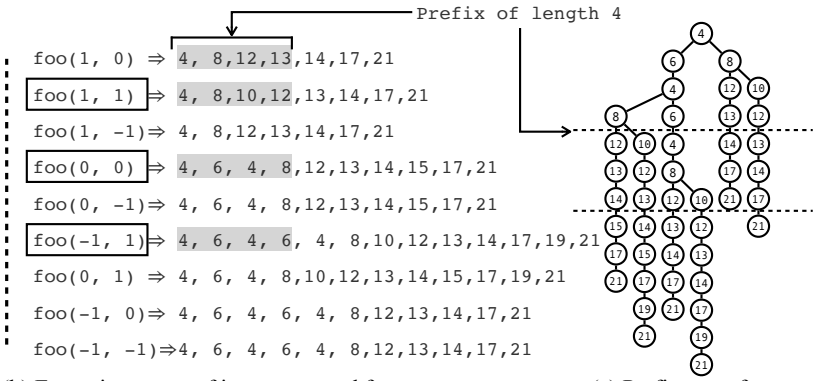
Significant research efforts have targeted this inefficiency, such as (1) *seed scheduling* [Lyu et al. 2019; Manès et al. 2020]: effectively select seed-to-mutate from the seed pool to prioritize seeds that are likely coverage-increasing or bug-triggering, (2) *tracing-cost reduction* [Nagy and Hicks 2019; Nagy et al. 2021; Zhou et al. 2020]: reduce the cost of coverage tracing with sophisticated designs, and (3) *new mutations* [Lemieux and Sen 2018; She et al. 2020]: perform targeted instead of random mutations to increase the likelihood of covering new regions. However, they still need

Authors' addresses: Shaohua Li, shaohua.li@inf.ethz.ch, ETH Zurich, Switzerland; Zhendong Su, zhendong.su@inf.ethz.ch, ETH Zurich, Switzerland.

2475-1421/2023/4-ART75
https://doi.org/10.1145/3586027

```
1  int foo(int a, int b)
2  {
3    int ret = 0;
4    while ( a < 1)
5    {
6      a++, ret++;
7    }
8    if (b == 1)
9    {
10     a--;
11   }
12   if (a + b >= 0) {
13     ret++;
14     if (ret == 2)
15       ret++;
16   }
17   if (a==0 && b==1
18       && ret==3) {
19     ret = 0;
20   }
21   return ret;
22 }
```

(a) A constructed code snippet.



(b) Execution traces of inputs mutated from `foo(1, 0)`. (c) Prefix tree of traces.

Fig. 1. Illustrative example.

to fully execute a large amount of non-coverage-increasing tests. Thus, avoiding executing non-coverage-increasing tests can significantly improve fuzzing efficiency. The key challenge is how to effectively make this determination at a low cost. Indeed, if we can decide whether or not a test increases coverage at a lower cost than a full test execution, we can reduce execution overhead and thus improve fuzzing efficiency.

To this end, this paper introduces prefix-guided execution (PGE), a novel solution to the aforementioned challenge. PGE monitors the execution of a test and applies early termination when the execution prefix indicates that the test unlikely increases coverage. Generally, a program execution is a temporal transition sequence of program states. Code coverage metrics abstract over both program states and transitions. We refer to *full execution* as the complete temporal sequence, while *execution prefix* as the contiguous subsequence from the program entry. PGE is based on the hypothesis that whether a full execution leads to increased coverage may be effectively inferred from its prefix. Suppose that a short *execution prefix* suffices for accurately inferring whether the corresponding *full execution* increases code coverage, we can speed up fuzzing by focusing on only fully executing those coverage-increasing tests.

Figure 1 illustrates the intuition behind PGE with a constructed code snippet. Figure 1(a) shows the constructed function `foo`. To maximize code coverage, *e.g.*, line coverage, a fuzzer needs to generate tests to reach lines 6, 10, 13, 15, and 19, all of which rely on earlier program executions. For instance, reaching line 19 requires the satisfaction of three guards and the referred variables being

suitably used/updated by previous statements. Figure 1(b) lists execution traces of tests mutated from the seed `(1,0)` in a top-down manner. Rectangular frames highlight coverage-increasing tests. Out of the eight mutated tests, three increase line coverage. We also build the corresponding prefix tree for these traces in Figure 1(c). From this prefix tree, we can see that, to differentiate all *full execution* traces, prefixes of length 7 are needed (indicated by the second dotted line). To separate coarser-grained coverage-increasing traces, prefixes of length at most 4 are then sufficient (indicated by the first dotted line). Figure 1(b) highlights in grey the first occurrences of unique prefixes of length 4. We can find that these unique prefixes cover all interesting tests, namely the seed and coverage-increasing tests. Suppose that a fuzzer monitors execution prefixes (with a fixed length of 4) and terminates an execution immediately whenever its prefix has occurred before. Fuzzing `foo` with tests in Figure 1(b) would result in 4 full executions on prefix-unique tests and 5 partial executions on the rest. The execution overhead is thus reduced on 5 out of the 9 tests. In practice, a large fraction of tests generated by fuzzers is neither coverage-increasing nor prefix-interesting. Guiding executions with such prefixes in fuzzing can potentially lead to significant cost reduction.

At a high level, for a CGF-style fuzzer, PGE works as a replacement for the fuzzer's execution engine. Instead of fully executing all tests, PGE selects prefix-interesting ones to fully execute them and discards all the others. A proper prefix length is crucial for the performance of PGE. However, for a general target, it is challenging to determine a prefix length that can identify coverage-increasing tests while being as small as possible. Utilizing static/symbolic analysis to reason about constraints between different program locations and decide a proper prefix length, although plausible in theory, is difficult to (1) scale to complex real-word code (which is why fuzzing is much more widely adopted) and (2) handle low-level code (*e.g.*, binaries) for settings where source code is unavailable. To tackle this challenge, we propose a sampling-based search algorithm to dynamically infer prefix lengths with low overhead. Our algorithm simulates the current fuzzing loop and finds the prefix length that can recall a desired amount of interesting tests.

As Section 2 will show, a relatively short execution prefix can help identify a large proportion of coverage-increasing tests. Since finding bugs is the ultimate goal of fuzzers, we will also show that execution prefixes can also effectively identify bug-triggering tests.

In summary, this paper makes the following main contributions:

- It studies the correlation between execution prefixes and the coverage increasingness of tests by quantifying their strong connection across nine real-world projects.
- It proposes the novel, general technique of prefix-guided execution (PGE) to speed up and improve fuzzing by early terminating executions that unlikely increase coverage.
- It presents a simple prefix length search algorithm for finding a proper prefix length to effectively guide PGE.
- It realizes PGE in AFL++-PGE, a prototype on top of AFL++, and extensively evaluates AFL++-PGE to demonstrate its utility in terms of fuzzing cost reduction, coverage increasing, and bug detection.

2 OBSERVATIONS ON CGF

This section introduces several key observations on coverage-guided fuzzing (CGF) that PGE builds upon. Without loss of generality, given a target binary and an initial seed pool, the high-level workflow of a fuzzer is as follows:

(1) **Seed selection.** Select one seed from the seed pool according to predefined strategies.

(2) **Test generation.** Mutate the seed to generate a large number of tests.

Table 1. Rates of Coverage-increasing Tests Out of All Tests Generated in One Hour By AFL++.

libpng	libsndfile	libtiff	libxml2	lua	openssl	php	poppler	sqlite3	**avg.**
0.03%	0.05%	0.11%	0.21%	0.24%	1.00%	0.25%	2.12%	0.11%	**0.46**%

(3) **Execution and feedback collection.** Execute each of the tests on the target binary and collect coverage feedback, queue a test to the seed pool if it increases coverage, and report it when it causes an execution failure, *e.g.*, crash.

(4) **Go to step 1 and repeat.**

The workflow shows that coverage-guided fuzzers work in a loop. In each loop, the selected seed will be mutated and blindly executed a tremendous amount of times to filter out those coverage-increasing ones. However, as has been reported by previous work [Nagy and Hicks 2019], the overwhelming majority of test cases are non-coverage-increasing. Although executing the target binary on them wastes most of the allocated resources, fuzzers still have to do so as it is by now the only way to understand whether a test increases coverage.

However, as we will show in this section, it is possible to accurately infer coverage-increasing property without full test execution—partial execution suffices in most cases. We use 21 programs from nine real-world projects in Magma benchmark [Hazimeh et al. 2020] to demonstrate the key observations that our design relies on. We averaged the results of programs from the same project. We choose, as our target fuzzer, the most popular coverage-guided fuzzer AFL++ [Fioraldi et al. 2020] on which many fuzzers are built. All experiments are run with the same setup as that for our later performance evaluation.

Observation 1: Only a tiny fraction of tests are coverage-increasing.

We run AFL++ on each program for an hour. All experiments are repeated 12 times with different initial seeds. We log the total number of generated tests during fuzzing. The number of coverage-increasing tests are learned by counting the new seeds that AFL++ appends into the seed pool. In AFL++, a test is interesting when it reached new edges or significantly increased edge counts. We treat both cases as coverage-increasing. Table 1 shows the averaged rates of coverage-increasing tests during one hour of fuzzing. On seven out of nine projects, less than 1% tests generated by AFL++ increased coverage. On average, AFL++ has 0.46% coverage-increasing tests out of all tests in each one hour trial. That means during the first hour of fuzzing, averagely only 4 out of 1000 tests are actually interesting and worth executing. If we are able to choose only those interesting ones, the execution cost on the remaining large number of tests can be saved.

Observation 2: Execution prefixes correlate highly with a test's coverage increasingness.

Mainstream coverage-guided fuzzers use basic block edges as their coverage metrics (26 out of 27 as reported in [Nagy et al. 2021]), so our study focuses on edge coverage as well. For a coverage-guided fuzzer, we denote an execution trace T^k as an ordered temporal sequence $T^k = \langle e_0^k, e_1^k, \ldots, e_{n-1}^k \rangle$, where $e_i^k, i \in [0, n)$ is the i-th edge this execution accessed. Tracing all temporal execution traces during fuzzing is extremely costly. Instead, AFL++ uses a global coverage bitmap

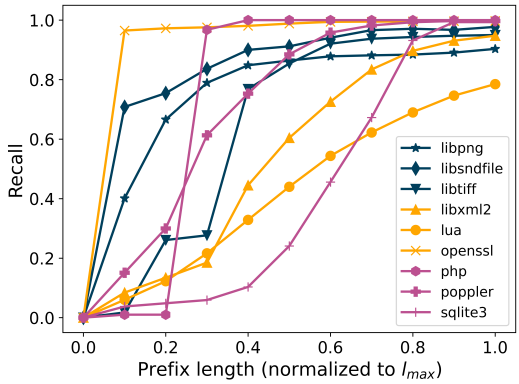


Fig. 2. Recall of coverage-increasing executions from interesting execution prefixes.

with counts to store the accessed edges such that an execution is identified by $E^k = multiset(T^k) = \{e_0^k, e_1^k, \ldots, e_{n-1}^k\}$[1], where the item ordering is ignored and duplicated edges are included. Before introducing our observation, we first define the notion of *execution prefix* used in this work.

Definition 2.1 (Execution Prefix). Given an execution trace T^k of length n and predefined prefix length $l \leq n$, the *execution prefix* of T^k is defined as $\Pi^k(l) = multiset(\langle e_0^k, e_1^k, \ldots, e_{l-1}^k \rangle) = \{e_0^k, e_1^k, \ldots, e_{l-1}^k\}$.

This definition shows that an execution prefix $\Pi^k(l)$ is a subset of the corresponding full execution E^k. This feature allows us to reuse the bitmap structure in AFL++ and obtain $\Pi^k(l)$ by terminating the execution when accessing the l-th edge (See technical details in Section 3.2). AFL++ considers a test *interesting* when it increases coverage. Since our goal here is to discover the connection between execution prefixes and such interestingness of full executions, we define next the notion of *interesting execution prefix*.

Definition 2.2 (Interesting Execution Prefix). Given a predefined prefix length l, m seen executions $\mathcal{E} = \{E^0, E^1, \ldots, E^{m-1}\}$, and their execution prefixes $\mathcal{P} = \{\Pi^0(l), \Pi^1(l), \ldots, \Pi^{m-1}(l)\}$. The subsequent execution prefix $\Pi^m(l)$ of E^m is interesting iff $\Pi^m(l) \notin \mathcal{P}$.

This definition shows that when determining whether an execution prefix $\Pi^m(l)$ is interesting or not, each seen execution prefix in $\mathcal{P}$ is taken into account. In our implementation, we cached the hash values of all execution prefixes for such lookups (Technical details in Section 3.3). Suppose that an execution E^m is interesting *w.r.t.* code coverage implies its prefix $\Pi^m(l)$ is also interesting *w.r.t.* Definition 2.2. We could then use execution prefixes to filter out all uninteresting tests without fully executing them. Since it takes less time to obtain execution prefixes than full executions, this strategy can potentially boost fuzzers' efficiency.

We now empirically validate this assumption to understand how commonly it holds in practice. Execution prefixes can be obtained with a modified AFL++, which terminates an execution when it reaches a predefined prefix length limit. Technical details will be shown in Section 3. Experiments on each program are done 12 rounds with different initial seeds. For each round of experiment on a program, we choose one seed and follow the steps below:

(1) Run AFL++ on one seed with a one-hour timeout and collect all generated tests. Identify the length of each execution and label all coverage-increasing executions.

[1] A *multiset* contains duplicated items while item ordering is ignored (https://en.wikipedia.org/wiki/Multiset).

Table 2. The Number of Selected Tests Out of All Bug-triggering Tests by Different Metrics.

	libpng	libsndfile	libtiff	libxml2	lua	openssl	php	poppler	sqlite3
Total	62	32	252	921	6	18	34	212	68
Coverage	8	6	10	377	2	4	5	52	14
Pattern	62	32	250	900	4	18	34	211	66

(2) Set the max prefix length l_{max} as the average length of the collected executions. In principle, as long as setting $1.0 \cdot l_{max}$ as the prefix length can retrieve nearly all interesting tests, the selected metric for l_{max} is eligible. Our preliminary experiments show that setting both mean and median prefix lengths as l_{max} are qualified. In our implementation, we choose to use the mean value because, unlike median, it does not need to track all intermediate values.

(3) Select 10 evenly distributed prefix lengths $l \in \{0.1 \cdot l_{max}, 0.2 \cdot l_{max}, \ldots, 1.0 \cdot l_{max}\}$. For each prefix length, rerun AFL++ on the same set of tests to collect their execution prefixes. Identify all interesting execution prefixes.

(4) Out of all coverage-increasing executions, count how many of them also have interesting execution prefixes, *i.e.*, the recall of coverage-increasing executions by interesting prefixes.

Figure 2 shows the average recall of coverage-increasing executions by interesting execution prefixes. The top six curves demonstrate that for these six projects, recall rates grow exponentially with the increase of prefix length. Short prefixes are less effective on `lua` and `sqlite3`, but longer prefixes still do. This experimental result reveals that a short execution prefix suffices to locate most of the coverage-increasing tests. On six out of nine projects, achieving 70% recall on average needs prefixes $\leq \frac{4}{10}$ in length of the full executions, and $\leq \frac{6}{10}$ for `libxml2`. Suppose that the prefix length to achieve a high recall is known to a fuzzer, it can partially execute most tests, while only fully executing those with interesting prefixes.

Observation 3: Not all bug-triggering tests are coverage-increasing.

For all tests mutated from one seed, suppose that an ideal fuzzer is able to identify coverage-increasing ones and only executes them. This fuzzer would achieve the same code coverage as executing them all. However, finding bugs is the ultimate goal of a fuzzer. We cannot guarantee that this ideal fuzzer can find the same number of bugs unless bug-triggering tests are also coverage-increasing.

To find out if bug-triggering tests could also increase coverage, we run AFL++ on the same set of programs for 48 hours, then collect all coverage-increasing tests (placed in "queue" directory) as well as unique crash-triggering tests (placed in "crashes" directory). We then rerun AFL++ on all these tests in order according to their creation timestamps to simulate the fuzzing process and check if a bug-triggering test increased code coverage at their creation time.

The "Total" row in Table 2 lists the average number of bug-triggering tests found in each project. Of those, the "Coverage" row shows the number of coverage-increasing tests. We can see that only a small fraction of bug-triggering tests increased code coverage. In this case, executing only coverage-increasing tests during fuzzing may miss many bugs. To address this issue, we need to relax the restrictions on interesting executions, which should contain not only coverage-increasing

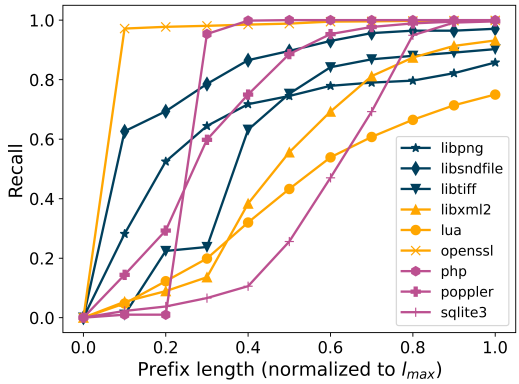


Fig. 3. Recall of pattern-interesting executions from interesting execution prefixes.

tests but bug-triggering tests as well. We choose to use the whole execution pattern, which can be viewed as an execution prefix with the maximum prefix length (*i.e.*, $l = n$ in Definition 2.1). Such patterns meet all of our requirements since (1) theoretically a coverage-increasing execution always implies its execution pattern being interesting, *i.e.*, never being seen; (2) empirically the "Pattern" row in Table 2 shows that most of the bug-triggering tests also have interesting patterns.

Recall Observation 2 discussed earlier, since we relaxed the scope on interesting executions, we now need to validate if execution prefixes still have a strong connection with new execution patterns. We run the same experiments as in Observation 2 but change target executions from coverage-increasing to pattern-interesting. Figure 3 shows the results. Execution prefixes keep the correlation tendency across projects but have relatively lower recall values. This is inevitable due to our relaxations but does not affect the overall effectiveness.

As will be detailed in Section 3.4, the whole execution pattern for each execution will only be used in the prefix search procedure of PGE. During prefix search, PGE evaluates each prefix by its recall capability of all pattern-interesting whole executions.

Summary: The above observations empirically show the relationships between coverage-increasing executions, bug-triggering tests, pattern-interesting full executions, and prefix-interesting executions. Figure 4 illustrates their relations pictorially. The target of our proposed PGE is to identify prefix-interesting tests and execute fully on them instead of all tests. Note that, this figure is only for conceptual illustration. Size ratios in the figure do not reflect their real values.

3 PREFIX-GUIDED EXECUTION

This section introduces technical details for prefix-guided execution (PGE) and how we augment a coverage-guided fuzzer with PGE.

3.1 Fuzzing with PGE

Figure 5 shows the high-level workflow of a coverage-guided fuzzer augmented with PGE. The three grey blocks highlight the key extensions that we make to the standard coverage-guided grey-box fuzzing. This new workflow illustrates that the fuzzing framework has not been altered and the PGE extension is generally applicable.

Prefix Execution is used to obtain the execution prefix of a test. Given a test and prefix length l, this module executes the target binary and terminates it whenever the execution visits l edges.

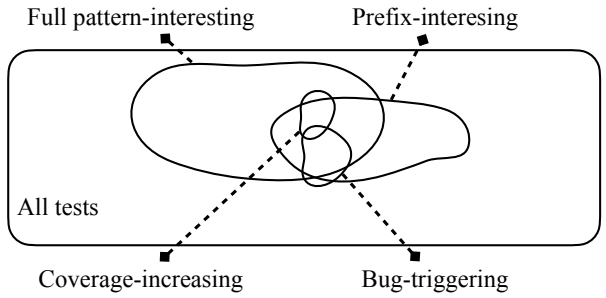


Fig. 4. Illustrative relations from observations.

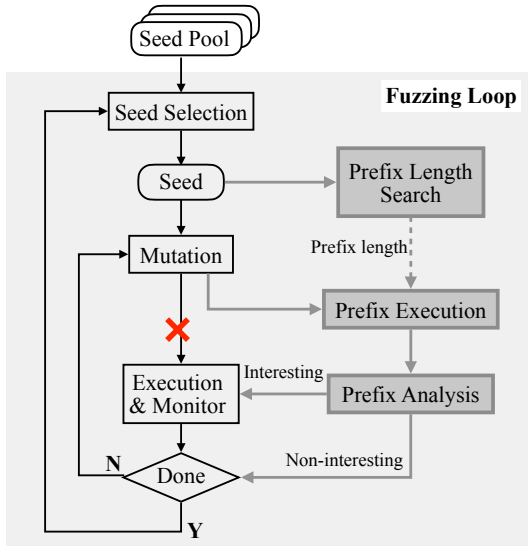


Fig. 5. Overview of a coverage-guided fuzzer augmented with prefix-guided execution.

As long as l is smaller than the actual execution length, such early termination reduces the time required for execution. This is a logic module that exists in the form of extra instrumentation code in the target binaries. (See details in Section 3.2.)

Prefix Analysis is for analyzing whether or not a given execution prefix is interesting. Interesting prefixes refer to those that have never been observed since the start of the current fuzzing loop. Only tests showing interesting prefixes will be fully executed while tests with non-interesting prefixes will be discarded (See details in Section 3.3).

Prefix Length Search estimates a proper prefix length for the current fuzzing loop. Recall the observations in Section 2, different prefix lengths correspond to different recall rates of pattern-interesting executions. When validating the observations, the concrete relationships between prefix lengths and recalls are from the post-processing of fuzzing loops. Now, when employing them in fuzzing, we need to learn such relationships before the start of a fuzzing process. We propose a

Algorithm 1: Fuzzing with PGE

```
   Input: Seeds 𝒮, Recall rate r .
1  while ¬Abort() do
2      s ← SelectSeed(𝒮)
3      l ← PrefixLengthSearch(s, r) // Find a prefix length l that can reach the recall r.
4      p ← AssignEnergy(s)
5      for i from 0 to p do
6          s′ ← Mutate(s)
7          Π^{s′}(l) ← PrefixExecution(s′, l) // Obtain the execution prefix of test s′.
8          if PrefixAnalysis(Π^{s′}(l)) == interesting then
9              E^{s′} ← FullExecution(s′) // If the execution prefix has never been seen.
10             if IsInteresting(E^{s′}) then
11                 add s′ to 𝒮
12         else
13             continue
```

sampling-based search algorithm to find an appropriate prefix length that can reach a given recall rate. (See details in Section 3.4).

Algorithm 1 shows an algorithmic sketch of how fuzzing with PGE works. The grey boxes highlight our extensions. The fuzzer is provided with a set of seeds $\mathcal{S}$ and a target recall rate r. During each fuzzing loop (the top *while* loop), a seed s is selected from $\mathcal{S}$ (line 2). All tests generated within the present fuzzing loop are derived from this seed. The fuzzer then searches for the smallest possible prefix length l that may reach the recall rate r, which is implemented in `PrefixLengthSearch` (line 3). An energy p is assigned to the seed, which represents the number of tests that will be generated according to the defined mutation operators (lines 4-6). For each new test s', the fuzzer partially executes the target binary, collects execution prefix $\Pi^{s'}(l)$ of length l (line 7), and goes to different branches:

(1) If $\Pi^{s'}(l)$ is interesting, *i.e.*, having never been seen in the current fuzzing loop, the fuzzer invokes the normal execution and monitor procedures to, *e.g.*, add s' to $\mathcal{S}$ if it increases code coverage, or cache s' if it results in a unique crash/hang. (lines 8-11)

(2) Otherwise, the fuzzer continues with the next test. (line 13)

According to the observations from Section 2, only a tiny fraction of tests explored in a fuzzing loop are interesting, *i.e.*, coverage-increasing or bug-triggering. If there were an ideal procedure that could select interesting tests without any execution overhead, a fuzzer, when equipped with such a procedure, could be dramatically more efficient. Such an ideal procedure certainly does not exist. `PrefixExecution` and `PrefixAnalysis` are designed to obtain a practical instance of such a procedure with partial execution overhead. For each test with an interesting execution prefix, the execution overhead increases from `FullExecution` to "`PrefixExecution` + `FullExecution`"; while for each test with an uninteresting execution prefix, the execution overhead decreases from

```
void
__sanitizer_cov_trace_pc_guard(uint32_t *guard) {

  __afl_area_ptr[*guard]++;

}
```

(a) Original AFL instrumentation code

```
void
__sanitizer_cov_trace_pc_guard(uint32_t *guard) {

  __afl_area_ptr[*guard]++;

  // Update the global prefix counter.
  __afl_prefix_cntr++;

  // Terminate the execution
  // when reaching the given prefix length.
  if (__afl_prefix_cntr == __afl_prefix_len)
    _exit(0);
}
```

(b) Extended AFL-PGE instrumentation code

Fig. 6. Original and extended AFL++-PGE instrumentation codes for edge coverage.

`FullExecution` to `PrefixExeuction`. The efficacy of prefix-guided execution critically depends on the proportion of interesting prefixes.

As will be shown in the evaluation, for most generated tests, AFL++ with PGE will execute the second branch above, *i.e.*, discarding the tests. A large proportion of uninteresting prefixes make the fuzzer discard most tests without fully executing them, thus improving fuzzing efficiency.

3.2 Prefix Execution

Given a prefix length l, `PrefixExecution` is able to terminate the target binary when it visits l edges. This is achieved by augmenting the standard coverage-tracing instrumentation. Next, we will use AFL++ as the target fuzzer to show how to support prefix execution in AFL++ instrumentation.

We use the "trace-pc-guard" mode in AFL++ for edge coverage tracing. This mode is backended by LLVM SanitizerCoverage [LLVM team 2021], which can insert calls to user-defined functions at the level of basic block edges. Figures 6 (a) and (b) show, respectively, the user-defined functions used in AFL++ and AFL++-PGE. In AFL++, edge coverage is reported by calling the function `__sanitizer_cov_trace_pc_guard()` with a unique identifier `guard` to increment the corresponding entry in `__afl_area_ptr` coverage map.

To count the number of edges visited, the extended instrumentation code increments a global counter `__afl_prefix_cntr` which is initialized to `0` at the start of an execution. The target prefix length is passed to `__afl_prefix_len` via a shared memory between the fuzzer and the target binary. When `__afl_prefix_cntr` reaches the limit specified by `__afl_prefix_len`, the ongoing execution will be terminated. At this moment, the coverage map `__afl_area_ptr` stores the execution prefix.

Note that the extended AFL++-PGE instrumentation code also supports full execution by setting `__afl_prefix_len=-1`, making the termination code unreachable. So, `PrefixExecution` and `FullExecution` in Alg. 1 in fact share the same instrumented target binary.

Algorithm 2: Prefix Length Search

Global Config: Sampling ratio sr.
Input: Seed s, Recall rate r.
Output: Prefix length l.

```
/* (1) Sampling tests with ratio sr                                          */
p ⟵ sr · AssignEnergy(s)
inputCache ⟵ [∅ ... ∅]_p
for i from 0 to p do
    inputCache[i] ⟵ Mutate(s)
/* (2) Identify if each full execution is interesting or not, and record average execution
   length.                                                                   */
arrFull, avgLen ⟵ BatchPrefixExecution(fullMap, inputCache, −1)
/* (3) Binary search the minimal prefix length that reaches target recall r. */
left ⟵ 0
right ⟵ avgLen
l′ ⟵ right, l ⟵ −1
while left < right do
    InitializeHashMap(prefixMap)
    arrPrefix, _ ⟵ BatchPrefixExecution(prefixMap, inputCache, l′)
    if CalculateRecall(arrFull, arrPrefix) ≥ r then
        right, l ⟵ l′
    else
        left ⟵ l′
    l′ ⟵ (left + right) / 2
return l
```

3.3 Prefix Analysis

We use a global hashmap `prefixMap` to record all prefixes that have been observed in the current fuzzing loop. For each new execution prefix, `PrefixAnalysis` first calculates its hash[2] value to index `prefixMap`: If the indexed entry is 1, it returns non-interesting; if the indexed entry is 0, it sets it to 1 and returns interesting. Note that, for every newly selected seed, `prefixMap` is emptied and rebuilt, which is due to the fact that the coverage map changes over time and the proper prefix length for the same seed may change accordingly. Because prefix lengths are not identical for different seeds, it is less meaningful to share `prefixMap` among them.We also use another such hashmap `fullMap` to record all pattern-interesting full executions. Since interesting full executions can be shared across seeds, different from `prefixMap`, the `fullMap` is only initialized in the beginning of fuzzing.

[2]We use the MurmurHash3 [Appleby 2016] hash function supported by AFL++.

Algorithm 3: BatchPrefixExecution

Input: hashMap, inputCache, prefix length l.

Output: An array of interestingness arrIntsg, average execution length avgLen.

```
1  p ⟵ LengthOf(inputCache)
2  arrIntsg ⟵ [0 ... 0]_p
3  avgLen ⟵ 0
4  for i from 0 to p do
5      Π^i(l), len ⟵ PrefixExecution(inputCache[i], l)
6      avgLen ⟵ avgLen + len
7      h_i ⟵ Hash(Π^i(l))
8      if hashMap[h_i] == 0 then
9          hashMap[h_i] ⟵ 1
10         arrIntsg[i] ⟵ 1
11     else
12         arrIntsg[i] ⟵ 0
13 avgLen ⟵ avgLen / p
14 return arrIntsg, avgLen
```

3.4 Prefix Length Search

To benefit from prefix-guided execution, a proper prefix length *w.r.t.* target recall r should be learned before fuzzing starts. The recall r describes the capability of a prefix in selecting interesting tests. A prefix length with higher recall implies that it is likely to select more interesting tests. However, for a recall r, the ground-truth minimal prefix length can only be learned by fully executing all candidate tests, thus it has no practical value.

To practically estimate a proper prefix length, we propose a sampling-based search algorithm. Instead of running all tests, the algorithm samples a tiny fraction, say 5%, runs them both fully and partially, and learns the recall capabilities of different prefix lengths. Alg. 2 details the prefix length search algorithm. It consists of three main stages:

(1) Same as common fuzzing procedure, an energy is assigned to the given seed but being reduced by a factor sr (line 1). Tests from p times mutations are then cached (lines 2-4).

(2) Before performing prefix length search, the interestingness of each full execution is learned via BatchPrefixExecution with prefix length $l = -1$ (recall `__afl_prefix_len=-1` in Section 3.2). The hashmap `fullMap` records hashes of all visited executions. `arrFull` is a binary array of size p, where `arrFull`$[i] = 1$ indicates `inputCache`$[i]$ being interesting. `avgLen` is the average execution length.

(3) From 0 to `avgLen`, we binary search for the minimal prefix length that reaches the target recall r. During each round of search, `prefixMap` is initialized first. With the same prefix length l', we learn the interestingness of each execution prefix via BatchPrefixExecution. The binary array `arrPrefix` stores such information just as `arrFull`. Function CalculateRecall calculates the recall rate of l'. The final prefix length l will be updated to l' when its recall rate is no less than r.

Function BatchPrefixExecution is described in Alg. 3. For each of the tests (line 4), it first gets the execution prefix of length l (full execution when $l = -1$). The hash of the execution prefix is calculated (line 7) to index the given hashmap (line 8). If the indexed entry is 0, *i.e.*, this is a new prefix, mark it as seen (line 9) and set `arrIntsg`$[i]$ to 1 (line 10). Otherwise, set `arrIntsg`$[i]$ to 0, *i.e.*, non-interesting (line 12). The variable `avgLen` accumulates execution length (line 6) and reports the average value finally (line 13). The returned array `arrIntsg` records whether execution prefixes of tests are interesting or not.

Function CalculateRecall calculates the recall as follows:

$$r = \frac{\sum_{i=0}^{p-1}(\texttt{arrFull}[i] \wedge \texttt{arrPrefix}[i])}{\sum_{i=0}^{p-1}\texttt{arrFull}[i]}.$$

Intuitively, this formula calculates to what percentage the 1's in `arrFull` are also marked as 1 in `arrPrefix`.

4 EVALUATION

This section details our extensive evaluation on PGE to demonstrate its effectiveness in improving fuzzing performance. Our evaluation will answer the seven research questions as shown below. The first three RQs are module-only evaluations on the key component PrefixLengthSearch, which adds extra fuzzing overhead. The rest of the RQs focus on PGE's overall performance.

RQ1. Accuracy of prefix length estimation at different sampling ratios,

RQ2. Overhead of the PrefixLengthSearch module,

RQ3. Distributions of prefix length on different recall settings,

RQ4. Early terminated tests as a percentage of all tests,

RQ5. Ratio of executing interesting tests to all full executions,

RQ6. Effectiveness of fuzzing in terms of bug finding, and

RQ7. Effectiveness of fuzzing in terms of code coverage.

Experimental Setup. We used AFL++ 4.01c [Fioraldi et al. 2022], the latest version at the time of writing, as the reference fuzzer to study the benefits of PGE. We refer to our augmented AFL++ as AFL++-PGE. We chose to use the most recent fuzzing benchmark Magma v1.2 [Hazimeh et al. 2020], the latest version at the time of writing. Magma consists of 21 programs from nine popular real-world projects, which were selected for their diverse functionalities. Table 3 details these targets. Due to our implementation limitation, we cannot support persistent fuzzing and thus all persistent targets such as pdf_fuzzer are using AFL driver with $N = 1$.

To test PGE's effectiveness against other fuzzing throughput boosters, we compare it with HeXcite [Nagy et al. 2021], the state-of-the-art coverage-guided tracer for coverage-guided fuzzers. HeXcite improves fuzzing throughput by restricting instrumentation overhead to coverage-increasing tests only. It extends Untracer [Nagy and Hicks 2019] with the support of edge coverage.

We performed our experiments on two servers, each equipped with an AMD Ryzen Threadripper 3990X 64-Core 2.9GHz CPU and 256 GB RAM, and running Ubuntu 20.04.3 LTS. Following Klees *et al.*'s [Klees et al. 2018] standard we performed each fuzzing campaign for 48 hours and repeated it 12 times.

Table 3. Magma targets.

Target	Drivers	Version	File type
libpng	libpng_read_fuzzer	1.6.38	PNG
libsndfile	sndfile_fuzzer	1.0.31	Audio
libtiff	tiff_read_rgba_fuzzer, tiffcp	4.3.0	TIFF
libxml2	xml_read_memory_fuzzer, xmllint	2.9.12	XML
lua	lua	5.4.3	LUA
openssl	asn1, asn1parse, bignum, server, client, x509	3.0.0	*Binary blobs*
php	json, exif, parser, unserialize	8.1.0alpha3	*Various*
poppler	pdf_fuzzer, pdfimages, pdftoppm	21.07.0	PDF
sqlite3	sqlite3_fuzz	3.37.0	SQL

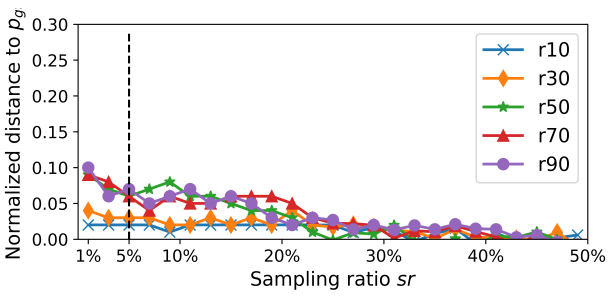


Fig. 7. Estimation accuracy of different sampling ratios.

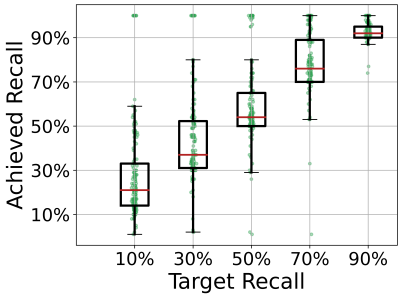


Fig. 8. Achieved recall vs. target recall with sampling ratio 5%.

4.1 RQ1: Accuracy of Prefix Length Estimation at Different Sampling Ratios.

Given a target recall, PrefixLengthSearch simulates fuzzing with sampling data points to estimate an appropriate prefix length that can achieve it. To understand what sampling ratio is required for an accurate estimate, for each program and a target recall, we

(1) randomly select a seed,
(2) set the sampling ratio $sr = 100\%$ to get the ground truth prefix p_{gt},
(3) set sr again from 1% to 50% with step 2% and estimate the prefix p_i, and
(4) for each p_i, calculate its normalized distance to p_{gt} with $d_i = \frac{|p_i - p_{gt}|}{p_{gt}}$.

We select 6 recalls spanning evenly from 0% to 100%, namely 10%, 30%, 50%, 70%, and 90%. Each experiment is repeated 12 times and we report the averaged results across programs in Fig. 7. A higher sampling ratio indicates a more precise approximation of prefix length but a higher overhead due to the increased number of tests. When $sr > 5\%$ the distance to the ground truth prefix decreases slowly for all recalls. For example, extending sr from 5% to 20% only reduces the distance in less than $0.05 \cdot p_{gt}$. Since higher sr indicates higher overhead, we conclude that $sr = 5\%$ is a good balance between the estimation accuracy and overhead. In our implementation of AFL++-PGE, we set $sr = 5\%$ as default.

In order to understand whether or not PGE can maintain target recalls with sampling rate 5%, for each target recall we reuse the above experiment meta-data as follows:

(1) estimate the prefix length p with 5% sampled executions,

(2) for all executions, count the number of pattern-interesting full executions N_{full},

(3) for all executions, collect their prefixes of length p and then count the number of interesting prefixes N_{pre},

(4) and calculate the true achieved recall rate $\frac{N_{pre}}{N_{full}}$.

Fig. 8 shows the distribution of achieved recalls on all programs. The X-axis refers to the target recall set for PGE and the Y-axis is the achieved recall. Each box shows the distribution of achieved recalls on all programs. We can find that for every target recall, PGE with 5% sampling rate can always achieve it in more than 75% of cases. The short length of each box indicates that the achieved recalls are concentrated around the target recalls. Overall, PGE with 5% sampling rate has successfully searched for proper prefixes.

4.2 RQ2: Overhead of Prefix Length Search

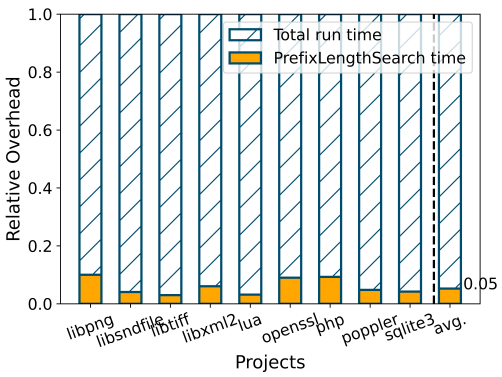


Fig. 9. Per-program relative overhead of PrefixLengthSearch in 48h.

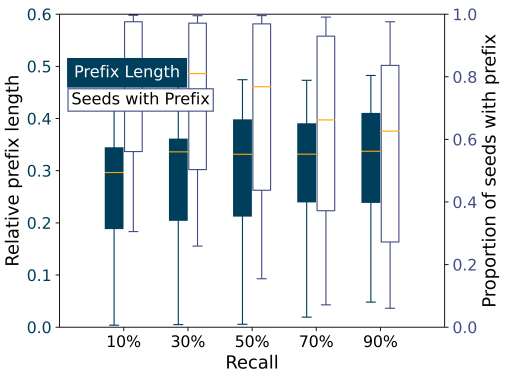


Fig. 10. Distributions of prefix lengths and proportion of seeds with prefix.

Since *prefix length search*, *i.e.*, function PrefixLengthSearch, brings extra overhead to the fuzzing loops, we first need to understand its cost in time against the overall fuzzing overhead. If AFL++-PGE took much time in searching for a prefix length, even though it could benefit from prefix execution, its overall performance would still be severely affected. To answer this question, we recorded time spent executing PrefixLengthSearch during the fuzzing campaigns.

AFL++-PGE needs an extra input parameter, recall rate r, for guiding PrefixLengthSearch. We select 6 different recalls spanning evenly from 0% to 100%, namely $r = 10\%, 30\%, 50\%, 70\%,$ and 90%.

For each target program, we ran AFL++-PGE with different recalls and averaged their PrefixLengthSearch time. The final averaged overhead is reported in Fig. 9. The last bar in Fig. 9 shows that during 48 hours of fuzzing campaign, the average time spent executing PrefixLengthSearch is less than 5%, approximately two hours. For most programs, PrefixLengthSearch overhead is negligible. Projects `libpng`, `openssl` and `php` have relatively larger overhead but all below 10%. Overall, the extra overhead from PrefixLengthSearch did not hinder AFL++-PGE's efficiency.

4.3 RQ3: Distributions of Prefix Length on Different Recall Settings

Prefix length is crucial for the efficiency of PrefixExecution. Intuitively, shorter prefix length means a faster PrefixExecution. Understanding what the distributions of prefix length would be on different recall settings is thus essential to understanding the overall performance of AFL++-PGE.

For each trial on a program, we logged the searched prefix lengths and execution lengths. Note that, for some seeds, PrefixLengthSearch may return -1 meaning that there is no effective prefix length and the fuzzer should continue with the original FullExecution for this seed. This is because sometimes even the maximal prefix (*i.e.*, average full execution length) cannot achieve the target recall. For seeds with effective prefix length, we average their relative prefix lengths *w.r.t.* execution lengths on the program-level granularity. Distributions of relative prefix lengths are shown as the solid boxes in Fig. 10. As expected, a higher recall needs larger prefix lengths. We also show the proportion of seeds with effective prefix length as the hollow boxes in Fig. 10. Inversely, the number of seeds with effective prefix length is less for a higher recall. For instance, when targeting the 50% recall rate, on average more than 45% seeds can be run with PrefixExecution, among which the searched prefix lengths are approximately one-third to full execution lengths. Although smaller recall rates come with shorter prefix lengths, they by design have higher probabilities of missing interesting tests. Section 4.6 and 4.7 will show the trade-off.

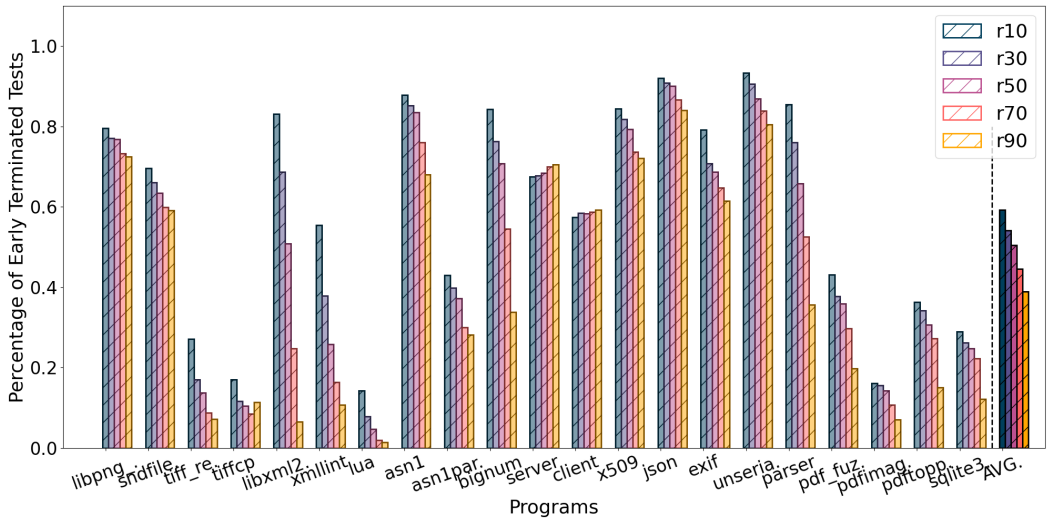


Fig. 11. Percentage of early terminated executions in AFL++-PGEs.

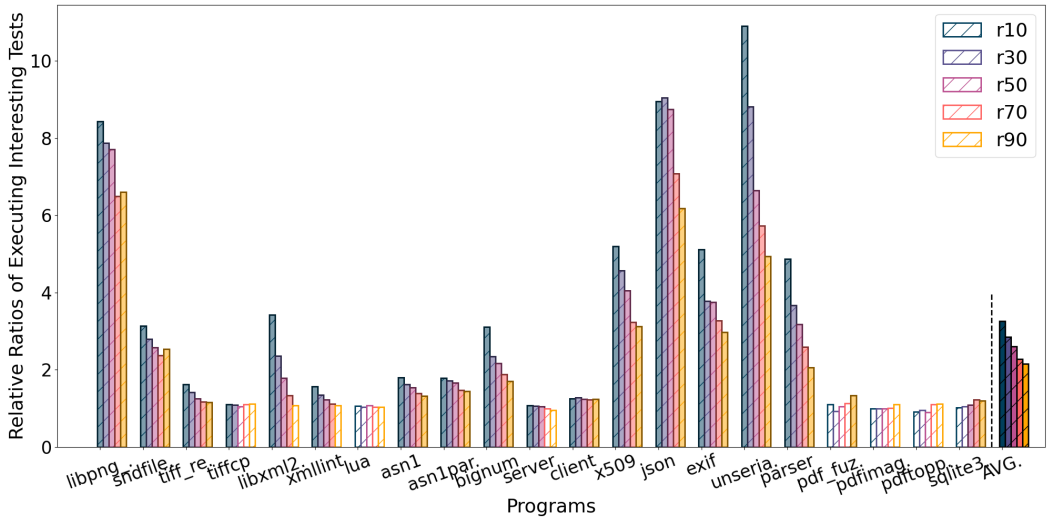


Fig. 12. AFL++-PGEs' ratios of executing coverage-increasing tests to all full executions relative to AFL++. Each ratio is calculated by #coverage-interesting tests / #full executions. Each solid bar indicates that the corresponding AFL++-PGE has a significantly higher ratio than AFL++ (*i.e.*, Mann-Whitney U test $p < 0.05$).

4.4 RQ4: Early Terminated Tests as A Percentage of All Tests

With prefix-guided execution, a fuzzer saves execution time by only partially executing tests. We now aim to answer how many tests would be early terminated by AFL++-PGEs. An early terminated test means that it has been only partially executed. Fig. 11 shows the mean ratios of early terminated executions per benchmark. We can observe that lower recall settings have a relatively higher percentage of early terminated tests. Intuitively, a lower recall allows PGE to search for a shorter prefix, which is less sensitive compared to longer prefixes and thus results in more early terminated tests. AFL++-PGE with recall 10% early terminated more than 90% tests on programs "asn1", "json", and "unserialize". As will be shown in Section 4.6 and 4.7, such a low recall, however, enables PGE to discover more bugs and achieve higher coverage on these benchmarks. Averagely, AFL++-PGEs with different recalls have 40% to 60% of tests being early terminated. For the fully executed tests, the majority of them are from seeds where PGE does not find effective prefixes.

4.5 RQ5: Ratio of Executing Interesting Tests to All Full Executions

We have shown in Section 2 that most of the fuzzer-generated tests are not interesting, *i.e.*, in the first hour of fuzzing, only 0.46% tests are coverage-interesting. This observation motivated our PGE design. Ideally, PGE should be more concentrated than a vanilla fuzzer on executing interesting tests. To measure such concentration, we define the interesting ratio as follows:

$$r = \frac{\text{\#coverage-increasing tests}}{\text{\#fully executed tests}}.$$

For AFL++, all tests are fully executed. In AFL++-PGEs, only partial tests are fully executed since many of the generated tests are early terminated by PGE. We calculate the interesting ratios for each fuzzer on each program and average the ratios across all trials. We use Mann-Whitney U-*test* to measure the statistical significance compared to AFL++. Fig. 12 shows the mean relative ratios

of AFL++-PGE to AFL++ on each program. We can see that on 18 out of 21 programs, at least one of the AFL++-PGEs has a significantly higher interesting ratio than the vanilla AFL++. On 12 out of 21 programs, all AFL++-PGEs have significantly higher interesting ratios. For programs "libpng_read_fuzzer", "x509", "json", "exif", "unserialize", and "parser", AFL-PGEs have 3x to 11x higher interesting ratios.

Note that, the interesting ratios do not directly reflect the effectiveness of the fuzzers. For example, although AFL++-PGE-r90 has a relatively low ratio on "sqlite3", as shown in Section 4.6, AFL++-PGE-r90 can find 3 bugs on it while the vanilla AFL++ can only find 1 bug. The main reason for the relatively low interesting ratios of AFL++-PGE on some programs is that after 48 hours of fuzzing, AFL++-PGEs saturate on some programs, which leads to a significant increase in the total number of full executions, while the increase in coverage is subtle due to saturation, resulting in a less significant interesting ratio r. As shown in Section 4.3, PGE can not always find effective prefix. For these programs, PGEs have turned back to the normal fuzzing procedure on some seeds, which results in the same ratios on these seeds.

4.6 RQ6: Bug-Finding Evaluation

The ability to discover bugs is the golden metric for fuzzing performance. We evaluate the bug detection capability of each fuzzer with *bug survival time*, which is proposed in the Magma. This metric uses the time required to trigger a bug to measure bug-finding speed. Due to the highly-stochastic nature of fuzzing, the time-to-bug might differ dramatically across repeated attempts. To mitigate the high variations in bug discovery time, Magma recommends the use of *survival analysis* to infer how long a bug "survives" in a fuzzing campaign. It adopts Wagner's approach [Wagner 2017] and uses the Kaplan-Meier estimator [Kaplan and Meier 1958] to estimate a bug's survival time, *i.e.*, staying undiscovered, within a given time (48 hours in our evaluation). A smaller survival time means better fuzzing performance. We also use the *log-rank test* [Herrera et al. 2021; Mantel et al. 1966] to statistically compare bug survival times. The *log-rank test*' p-value<0.05 implies statistical significance. Table 4 summarizes the bugs found in Magma and the mean survival time. We omit programs when none of the fuzzers was able to find any bugs in them and bugs when all fuzzers were able to find them without statistical significance. Due to HeXcite's lack of support for the persistent mode that most programs use, we were only able to run it on five programs.

Versus AFL++: In total, AFL++ was able to find 25 bugs and AFL++-PGEs uniquely discovered 10 more. Specifically, AFL++-PGE with recall 10%, 30%, 50%, 70%, and 90% covered 4, 6, 5, 2, and 7 more bugs than vanilla AFL++. Of all these bugs[3], AFL++-PGEs were faster than AFL++ respectively on 8, 11, 9, 9, and 11 of them, and slower respectively on 5, 6, 4, 3, and 2 of them. Overall, AFL++-PGE-r90 was the best in terms of bug-finding ability.

An interesting fact is that a very low recall still has a strong ability in discovering bugs. For instance, AFL++-PGE with recall 30% triggered 29 bugs across all projects and uniquely discovered bugs "XML002" and "PDF014 (pdf_fuzzer)" that no other fuzzers did. Intuitively, a lower recall indicates that the searched prefix length has a higher probability of missing interesting tests. However, on one hand, we observed from Magma's intermediate logs that for nearly all bugs, if being triggered, would be triggered more than once. Suppose that for some bug, there are multiple triggering tests generated during fuzzing. Even if each single test has a low probability to be executed, the probability that this bug being triggered at least once is still high according to the chain rule [Schum 2001] in probability theory. But the triggering time might be delayed. For example, AFL++-PGE-r10 triggered "PHP009" much later than many others. On the other hand,

[3]When we compare AFL++-PGE-rXX with AFL++, only bugs triggered by at least one of these two fuzzers are included.

Table 4. Mean survival time in 48 hours. Bugs that have never been found are marked by "⊤". Survival times either statistically better than AFL++ (p-val < 0.05) or uniquely identified by AFL++-PGE are highlighted in green. "✗" means the program is incompatible with the fuzzer.

			AFL++-PGE					
Program	Bug	AFL++	r10 p-val	r30 p-val	r50 p-val	r70 p-val	r90 p-val	HeXcite p-val
libpng	PNG001	⊤	⊤ –	⊤ –	48h –	⊤ –	45h –	✗ –
	PNG007	31h	19h 0.08	6h **0.00**	8h **0.00**	10h **0.00**	9h **0.00**	✗ –
libsndfile	SND001	0.4h	0.2h 0.14	0.2h 0.43	0.3h 0.62	0.3h 0.58	0.4h 0.77	47h **0.00**
	SND005	0.9h	1h 0.60	2h 0.73	1.0h 0.54	0.5h **0.01**	0.7h 0.22	0.8h 0.48
	SND024	0.6h	0.4h **0.01**	0.2h **0.00**	0.2h **0.00**	0.3h **0.00**	0.4h **0.05**	⊤ –
libtiff-tiff_read_rgba	TIF002	38h	47h 0.05	⊤ –	⊤ –	45h 0.17	47h **0.04**	✗ –
	TIF008	42h	⊤ –	47h 0.23	⊤ –	⊤ –	⊤ –	✗ –
libtiff-tiffcp	TIF006	22h	32h 0.13	38h **0.02**	32h 0.21	39h **0.01**	36h 0.08	18h 0.81
	TIF007	0.1h	0.1h 0.45	0.1h 0.17	0.1h 0.64	0.0h 0.88	0.0h 0.40	32h **0.00**
	TIF014	2h	1h 0.22	2h **0.02**	2h 0.16	1h 0.29	2h 0.27	⊤ –
libxml2-xml_read_memory	XML001	1h	0.4h **0.00**	0.8h **0.01**	2h 0.22	1h 0.89	2h 0.47	✗ –
	XML009	2h	1h 0.13	3h 0.67	1h **0.02**	0.8h **0.01**	2h 0.77	✗ –
	XML012	⊤	⊤ –	⊤ –	48h –	⊤ –	48h –	✗ –
libxml2-xmllint	XML001	44h	⊤ –	48h **0.04**	47h 0.15	47h 0.29	45h 0.55	⊤ –
	XML002	⊤	⊤ –	46h –	⊤ –	⊤ –	⊤ –	⊤ –
	XML009	1.0h	3h **0.03**	2h 0.18	0.9h 0.86	2h 0.22	3h **0.03**	⊤ –
openssl-asn1	SSL001	⊤	10h –	5h –	4h –	10h –	4h –	✗ –
openssl-server	SSL002	0.2h	0.2h **0.00**	0.2h **0.00**	0.2h **0.00**	0.2h **0.00**	0.2h **0.00**	✗ –
	SSL020	45h	28h **0.02**	40h 0.32	41h 0.32	29h **0.02**	30h **0.04**	✗ –
php	PHP009	1h	13h **0.00**	14h **0.00**	6h **0.02**	10h **0.00**	3h 0.57	✗ –
poppler-pdfimages	PDF008	46h	⊤ –	44h 0.95	⊤ –	47h 0.95	45h 0.95	⊤ –
	PDF014	⊤	⊤ –	46h –	⊤ –	⊤ –	45h –	⊤ –
	PDF018	5h	36h **0.00**	27h **0.00**	25h **0.00**	27h **0.00**	15h 0.16	⊤ –
	PDF021	43h	40h 0.40	39h 0.57	40h 0.63	⊤ –	45h 0.93	⊤ –
poppler-pdftoppm	PDF010	2h	1h 0.11	1h 0.21	1h 0.17	0.8h **0.01**	1h 0.12	31h 0.10
	PDF011	45h	46h 0.95	46h 0.62	47h 0.95	44h 0.95	41h 0.55	37h **0.01**
	PDF018	11h	36h **0.00**	28h **0.02**	36h **0.02**	17h 0.71	18h 0.54	⊤ –
	PDF019	⊤	46h –	45h –	47h –	⊤ –	⊤ –	⊤ –
poppler-pdf_fuzzer	PDF011	⊤	46h –	⊤ –	⊤ –	⊤ –	44h –	✗ –
	PDF014	⊤	⊤ –	48h –	⊤ –	⊤ –	⊤ –	✗ –
	PDF018	11h	42h **0.00**	37h **0.00**	44h **0.00**	26h 0.08	21h 0.14	✗ –
	PDF021	46h	41h 0.32	⊤ –	47h 0.55	43h 0.86	44h 0.86	✗ –
sqlite3	SQL012	47h	⊤ –	⊤ –	47h 0.95	⊤ –	48h 0.95	✗ –
	SQL013	⊤	⊤ –	46h –	47h –	⊤ –	48h –	✗ –
	SQL020	⊤	45h –	44h –	⊤ –	41h –	46h –	✗ –

Section 4.4 has shown that AFL++-PGE-r10 has the largest fuzzing throughput, which allows it to explore some new paths within the time constraint.

Versus HeXcite: HeXcite has the fastest discovery speed on "PDF011 (pdftoppm)". For the rest of the 5 bugs it discovered, AFL++-PGE-r90 is faster on 3 of them. On the 5 programs it supports, it missed 8 bugs compared to AFL++ and AFL++-PGE-r90 found 9 more bugs than it with statistical significance. Intuitively, HexCite should be able to boost fuzzing efficiency since it improves overall throughput. However, it does not guarantee to trace all critical edges that are important to path exploration. As its empirical evaluation revealed, HexCite only tracks 89% of such edges, which inevitably limits its path-discovering capability. Moreover, hit counts coverage in HexCite is limited to loops only. Hit counts refer to edge execution frequencies. Due to its high overhead in tracking all hit counts, HexCite focuses only on loops, which further hinders the exhaustiveness of its exploration.

4.7 RQ7: Coverage Evaluation

Code coverage is a common and popular evaluation metric for fuzzing when ground-truth bugs are not available. Previous work [Klees et al. 2018; Li et al. 2021] argued that higher coverage does not necessarily indicate better bug-finding capability. Nevertheless, for the thoroughness of comparison, we report branch coverage achieved by each fuzzer. We measure each trial's edge coverage with `afl-showmap` utility and compute the average coverage across all trials. We use Mann-Whitney *U-test* to measure the statistical significance comparing to AFL++. Table 5 summarizes the coverage results, where statistically higher coverage than AFL++ (*i.e.*, p-value < 0.05) is highlighted in green.

Versus AFL++: As Table 5 shows, of the total 21 programs, AFL++-PGE with recall 10%, 30%, 50%, 70%, and 90% achieve statistically higher coverage than AFL++ respectively on 10, 10, 10, 10 and 11 of them, and the same on the left. Intuitively, a lower recall indicates that PGE would have a higher probability of missing interesting tests. However, all PGEs achieve nearly the same coverage at the end of 48h fuzzing. The main reason is that all PGEs have become saturated on these benchmarks. No matter how many new tests are explored, the overall improvements in coverage will be subtle. Thus, although PGE boosts the overall fuzzing speed substantially, it results in relatively small coverage improvements. For example, for recalls 10% and 30%, although they can explore more tests as shown in Section 4.4, their coverage performance does not surpass others. The highest recall 90% performed the best, which is aligned with its strongest bug-finding capability as shown in Section 4.6.

Higher coverage does not necessarily mean a stronger bug-finding capability. For instance, as shown in Section 4.6, AFL++-PGE-r50 statistically found more bugs than AFL++ in `libpng` and `sqlite3`, however, it achieves lower coverage than AFL++. As the bug-finding capability is the most important measure in fuzzing, PGE's significance in this measurement has shown its effectiveness in boosting fuzzing performance.

Versus HeXcite: For the five programs that HeXcite supports, it has statistically lower coverage than AFL++ on 4 of them and never outperforms AFL++. The best performing AFL++-PGE-r90 achieves higher coverage on 4 of them. Although HeXcite is able to improve fuzzing throughput by design, as has been analyzed in Section 4.6, it trades its efficacy for speed by discarding many instrumentations. As reported in HeXcite, it is more significant in black-box setting and achieves nearly equivalent or worse performance on grey-box settings. Our evaluation confirmed this fact.

Table 5. Edge coverage (%) achieved by fuzzers. Coverages statistically higher than AFL++ are highlighted in green. "✗" means the program is incompatible with the fuzzer.

			AFL++-PGE					
Project	Program	AFL++	r10 MWU	r30 MWU	r50 MWU	r70 MWU	r90 MWU	HeXcite MWU
libpng	libpng_read_fuzzer	24.6	24.3 0.77	24.6 0.58	24.3 0.76	24.6 0.45	24.4 0.67	✗ –
libsndfile	sndfile_fuzzer	22.2	22.1 0.94	22.1 0.89	22.2 0.61	22.2 0.57	22.3 0.22	17.3 **0.00**
libxml2	xml_read_memory	18.5	18.2 1.00	18.3 1.00	18.4 1.00	18.5 0.59	18.6 **0.00**	✗ –
	xmllint	16.3	15.9 1.00	16.1 1.00	16.2 1.00	16.2 1.00	16.2 0.96	11.3 **0.00**
lua	lua	81.5	80.6 0.93	80.9 0.94	81.6 0.27	81.4 0.56	81.3 0.60	✗ –
openssl	asn1	10.5	11.7 **0.00**	11.8 **0.00**	11.8 **0.00**	11.8 **0.00**	11.8 **0.00**	✗ –
	asn1parse	1.4	2.0 **0.00**	2.0 **0.00**	2.0 **0.00**	2.0 **0.00**	2.0 **0.00**	✗ –
	bignum	1.2	1.9 **0.00**	1.9 **0.00**	1.9 **0.00**	1.9 **0.00**	1.9 **0.00**	✗ –
	server	15.3	16.9 **0.00**	16.9 **0.00**	16.9 **0.00**	16.9 **0.00**	16.9 **0.00**	✗ –
	client	15.1	17.1 **0.00**	17.1 **0.00**	17.1 **0.00**	17.1 **0.00**	17.1 **0.00**	✗ –
	x509	11.6	12.4 **0.00**	12.4 **0.00**	12.4 **0.00**	12.4 **0.00**	12.4 **0.00**	✗ –
libtiff	tiff_read_rgba	36.0	32.6 1.00	34.0 1.00	34.1 1.00	34.7 1.00	34.9 1.00	✗ –
	tiffcp	41.0	39.8 1.00	40.1 1.00	39.6 1.00	40.4 0.89	40.4 0.91	41.0 **0.23**
php	json	0.7	1.5 **0.00**	1.5 **0.00**	1.5 **0.00**	1.5 **0.00**	1.5 **0.00**	✗ –
	exif	0.9	1.7 **0.00**	1.8 **0.00**	1.8 **0.00**	1.8 **0.00**	1.8 **0.00**	✗ –
	unserialize	1.0	1.8 **0.00**	1.8 **0.00**	1.8 **0.00**	1.8 **0.00**	1.8 **0.00**	✗ –
	parser	4.9	5.3 **0.00**	5.4 **0.00**	5.5 **0.00**	5.5 **0.00**	5.5 **0.00**	✗ –
poppler	pdf_fuzzer	38.8	38.3 1.00	38.4 1.00	38.7 0.86	38.7 0.59	38.7 0.63	✗ –
	pdfimages	45.7	44.8 1.00	45.1 1.00	45.2 1.00	45.5 0.93	45.7 0.22	36.7 **0.00**
	pdftoppm	39.2	38.6 1.00	38.7 1.00	39.0 0.90	39.0 0.90	39.0 0.83	31.8 **0.00**
sqlite3	sqlite3_fuzz	42.6	40.3 1.00	40.3 1.00	41.3 0.98	42.4 0.53	43.3 0.11	✗ –

4.8 Finding Bugs in Latest Applications

Evaluations on the Magma benchmark programs have already shown the superiority of AFL++-PGE. In order to test if AFL++-PGE is able to boost AFL++'s bug-finding capability on real-world applications, we selected 6 up-to-date programs with diverse types. All programs have been well-fuzzed in the fuzzing community [Böhme and Falk 2020; Chen et al. 2020a; Huang et al. 2020; Wang et al. 2020]. We used their official test suites as the initial seeds for `ffmpeg`, `objdump`, `readelf`, and `nm-new`. Seeds for the remaining two programs are from UNIFUZZ [Li et al. 2021]. We set recall for AFL++-PGE to 90% as suggested by our evaluations on Magma. All experiments were done on the same environments as discussed in Section 4 with a timeout of 48 hours. Table 6 reports the number of bugs found by each fuzzer. We manually inspected and triaged all unique crashes to identify unique bugs. In summary, AFL++-PGE-90 discovered more bugs than AFL++ in 4 out of the 6 programs. All bugs found by AFL++ were also discovered by AFL++-PGE-r90, which means that PGE did not miss any bugs in these real-world applications. AFL++-PGE-r90 in total discovered 28

Table 6. Bugs detected by AFL++ and AFL++-PGE-r90.

			AFL++			AFL++-PGE-r90		
Target	File Type	Version	Found	New	Fixed	Found	New	Fixed
ffmpeg	Video	4.4	1	0	0	1	0	0
cflow	C	1.6.92	0	0	0	1	0	0
mp42aac	MP4	1.6.0	2	0	0	4	2	0
objdump	Binary	2.36.1	8	8	8	10	10	9
readelf	Binary	2.36.1	5	5	5	5	5	5
nm-new	Binary	2.36.1	8	7	7	12	11	10
Total			24	20	20	**33**	**28**	**24**

previously unknown bugs, 40% more than AFL++. We reported these bugs to the developers and 24 of them have already been fixed. We can thus conclude that PGE significantly improves AFL++'s performance in finding bugs in real-world applications.

4.9 Discussion

Impact of recall in PGE. As can be learned from Sections 4.3 and 4.4, a smaller recall rate means a shorter prefix and higher execution speed, but also means a higher probability of missing interesting tests. As a result, we should avoid selecting a recall that is too small. On the other hand, experimental results in Section 4.7 and 4.6 showed that higher recall did not always mean more effective performance. For example, AFL++-PGE-r90 performed better than AFL++-PGE-r50 in both code coverage and bug survival time. Although AFL++-PGE-r10 and r30 did not outperform AFL++-PGE-r90 overall, they found additional bugs in `xmllint` and `pdftoppm`. In summary, it is a trade-off to balance execution speed and bug-triggering probability. We conclude that the higher execution speed of relatively low recall compensated its low bug-triggering probability. A proper target recall in PGE should be neither too large nor too small. Recall 90% is the best-performing one considering both of our measures and can be a good choice in practice.

Orthogonality of PGE to various CGF efforts. At a high level, PGE is a surrogate module for full executions and does not affect other parts of a coverage-guided fuzzer. Algorithm 1 in Section 3.1 has shown the algorithmic sketch for a coverage-guided grey-box fuzzer (CGF) and our PGE extensions to it. Various CGF efforts try to improve different aspects of the fuzzing process and share the same CGF workflow. For instance, seed scheduling schemes such as AFLFast [Böhme et al. 2017a] optimize the SelectSeed (line 2) and AssignEnergy (line 4) functions. Mutator scheduling methods such as MOPT [Lyu et al. 2019] improve the Mutate function. AFL++ [Fioraldi et al. 2020] integrates many advances into AFL while still maintaining the same CGF workflow. PGE does not alter any of these optimized functions, and only conditions full executions *w.r.t.* their prefixes. PGE is therefore orthogonal to these efforts. Our extensive evaluation has shown the significant performance improvement of PGE to AFL++. We believe integration of PGE to other fuzzers will boost their performance as well. When integrating PGE into a new fuzzer, the main efforts are to update two components coordinately: For `PrefixExecution`, if the new fuzzer does not use AFL-style instrumentation, one needs to add a global counter and a guard for prefix collection in

the instrumentation code; For `PrefixLengthSearch`, mutation strategies should be aligned with the new fuzzer.

Support of stateful fuzzing. It is crucial to find security vulnerabilities in stateful and persistent targets such as network services. Stateful fuzzing, however, is difficult. Vanilla CGF fuzzers like AFL++ are limited in their abilities to support stateful targets like network services. Since PGE is built atop AFL++, it inherits this limitation. There is some interesting work that attempts to address this limitation. One representative effort is AFLNet [Pham et al. 2020], which augments AFL to make it state-aware. In principle, PGE could be integrated into it since AFLNet treats a message sequence as an input seed in a normal fuzzing scenario while still maintaining state information. Early termination in AFLNet would not lose the target state.

Compatibility to other fuzzing boosting schemes. An interesting and orthogonal boosting approach for fuzzing is checkpoint-based resume [Song et al. 2020]. The key insight is that kernel fuzzers frequently execute similar test cases with same prefixes. To avoid redundant prefix executions, checkpoints are used to save states for hot prefixes. All future test cases with prefixes being cached can then resume from checkpoints. In principle, this scheme could be augmented with PGE. The augmented fuzzer can start collecting prefixes from a resumed checkpoint and terminate a subsequent execution when its prefix is uninteresting. Since this scheme is applied in kernel fuzzing and PGE is initially designed for application fuzzing, it remains challenging and unknown how PGE would perform on top of it.

Limitations and future work. PGE opens up a new and orthogonal research direction for improving fuzzing efficiency by early terminating executions. In our current design, we make use of the coverage information from executions as the indicator for early termination. Although it has shown a superior performance, our current PGE cannot use a hundred percent of recall since it would require a longer thus less effective prefix. This is mainly due to the limited expressiveness of coverage pattern as the indicator for early termination. It would be an exciting and interesting future work to explore other more expressive indicators such as data flows [Gan et al. 2020] and variable states [Fioraldi et al. 2021]. Such indicators may have the potential to predict an execution's coverage increasingness in a much earlier stage and thus need a shorter prefix.

5 THREATS TO VALIDITY

We here discuss potential threats to the validity of our results and conclusions.

Threats to external validity. One threat to external validity is the benchmarking programs we used for our evaluation. To reduce this threat, we chose the Magma benchmark, which consists of 21 programs from nine real-world projects. All of them have been widely used for evaluating fuzzers' performance in the fuzzing community. These programs were carefully selected by the Magma authors according to their diversity in functionality. Another threat to external validity is the randomness in fuzzing. Due to the highly stochastic nature of fuzzing, different trials on the same benchmark may differ significantly. To deal with this issue, we repeated all our experiments 12 times and used the log-rank test and Mann-Whitney U-*test* to draw statistically significant conclusions.

Threats to conclusion validity. This threat to validity relates to the reliability of the chosen measurements and the used statistical tests. We measured the significance of our PGE with bug-finding capability and edge coverage. Both measurements are widely used in the fuzzing community. For the adopted statistical tests, we followed many existing fuzzers.

6 RELATED WORK

Coverage-guided Greybox Fuzzing. Since the success of AFL, there is a large body of work that incorporates techniques such as taint analysis [Aschermann et al. 2019; Chen and Chen 2018; Chen et al. 2019; Mathis et al. 2020; Rawat et al. 2017], symbolic execution [Noller et al. 2018, 2020; Stephens et al. 2016; Wang et al. 2018; Yun et al. 2018], static analysis [Chen et al. 2020b; Li et al. 2017; Peng et al. 2018], and deep learning [Godefroid et al. 2017; Rajpal et al. 2017; She et al. 2019; Wang et al. 2017], to improve the performance of greybox fuzzing. However, all of them still have to fully execute all generated tests and, to the best of our knowledge, PGE is the first extension to greybox fuzzing for reducing execution overhead via early termination.

Improving Fuzzing Throughput. Boosting coverage collection improves fuzzing speed at tracing level. UnTracer [Nagy and Hicks 2019] and HeXcite [Nagy et al. 2021] use coverage-guided tracing to decrease time handling non-coverage-increasing tests. They utilize the fact that a large number of tests are non-coverage-increasing and only instrument code regions that have never been covered. Zeror [Zhou et al. 2020] uses a similar idea but with finer-grained coverage. INSTRIM [Hsu et al. 2018] utilizes the existence of common program structures to instrument a small fraction of basic blocks for reconstructing coverage. However, it becomes less useful after LLVM incorporates a similar but more efficient functionality in its PCGUARD mode [LLVM team 2021]. At the system level, Xu. *et al.* [Xu et al. 2017] implement three operating primitives specialized for fuzzing to solve the scalability bottlenecks in file I/O and system calls.

These research efforts are related to prefix-guided execution as they all target reducing overhead introduced by fuzzing itself. PGE tackles this problem from an orthogonal and new perspective.

Reachability Prediction in Targeted Greybox Fuzzing. There is a large body of work [Chen et al. 2018; Wüstholz and Christakis 2020; Zong et al. 2020] on guiding fuzzers toward target locations since AFLGo [Böhme et al. 2017b]. Among them, two are particularly related to PGE as they share a similar philosophy. FUZZGUARD [Zong et al. 2020] trains a deep learning model for each target location and predicts if each test can reach the target location without executing them. It hypothesizes that tests reaching the same target location have common syntactic patterns. Wüstholz. *et al.* [Wüstholz and Christakis 2020] present an online static look-ahead analysis to determine if an execution prefix can reach a target location. These two efforts both avoid unnecessary full executions. Their effectiveness is guaranteed by the high concentration of tests in targeted greybox fuzzing both syntactically and semantically. Although they have different application scenarios from PGE, it is nevertheless interesting future work to explore if deep learning and static analysis are applicable in reducing full executions for general fuzzing.

7 CONCLUSION

We have presented PGE, a novel technique for boosting greybox fuzzing's efficiency and bug detection. Our high-level insight is that partial test execution can help separate interesting and non-interesting tests, thus a fuzzer can terminate those non-interesting executions early for higher fuzzing throughput. We have empirically shown that most tests during fuzzing are non-interesting and execution prefixes can help select interesting tests. As a proof-of-concept, we have integrated PGE into AFL++. Our results show that AFL++-PGE improves not only AFL++'s performance, but more importantly, also its coverage-increasing and bug-finding capability. PGE is general and, in principle, can enhance any greybox fuzzer. This work provides a simple, effective realization, and motivates future explorations of this direction.

8 DATA-AVAILABILITY STATEMENT

The artifact is publicly available [Li and Su 2023]. This artifact consists of three main components: 1. AFL++-PGE, the fuzzer which we implemented PGE on top of AFL++. 2. All experimental data that can be used to reproduce all reported statistics in the paper. 3. Magma integration which one can run to get all our experimental data on fuzzers. The key data reported in the paper are in Section 2 (Observation) and Section 4 (Evaluation). We provide detailed instructions to reproduce these data, i.e., Table 1, 2, 4, 5, 6 and Figure 2, 3, 7, 8, 9, 10, 11, 12.

REFERENCES

Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. 2016. Announcing OSS-Fuzz: Continuous fuzzing for open source software. *Google Testing Blog* (2016).

Austin Appleby. 2016. MurmurHash3. https://github.com/aappleby/smhasher/wiki/MurmurHash3

Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *Network and Distributed System Security*, Vol. 19. 1–15.

Marcel Böhme and Brandon Falk. 2020. Fuzzing: On the exponential cost of vulnerability discovery. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 713–724.

Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017b. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2329–2344.

Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017a. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering* 45, 5 (2017), 489–506.

Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2095–2108.

Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.

Peng Chen, Jianzhong Liu, and Hao Chen. 2019. Matryoshka: fuzzing deeply nested branches. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 499–513.

Yaohui Chen, Mansour Ahmadi, Boyu Wang, Long Lu, et al. 2020a. MEUZZ: Smart Seed Scheduling for Hybrid Fuzzing. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. 77–92.

Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020b. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1580–1596.

Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. 2021. The Use of Likely Invariants as Feedback for Fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*. 2829–2846.

Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. {AFL++}: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*.

Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2022. *American Fuzzy Lop plus plus (AFL++)*. Retrieved August 1, 2022 from https://github.com/AFLplusplus/AFLplusplus/releases/tag/4.01c

Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data flow sensitive fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. 2577–2594.

Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 50–59.

Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 3, Article 49 (Dec. 2020), 29 pages. https://doi.org/10.1145/3428334

Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. 2021. Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 230–243.

Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. 2018. Instrim: Lightweight instrumentation for coverage-guided fuzzing. In *Symposium on Network and Distributed System Security, Workshop on Binary Analysis Research*.

Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. 2020. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1613–1627.

Edward L Kaplan and Paul Meier. 1958. Nonparametric estimation from incomplete observations. *Journal of the American statistical association* 53, 282 (1958), 457–481.

George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138.

Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 475–485.

Shaohua Li and Zhendong Su. 2023. Accelerating Fuzzing through Prefix-Guided Execution. https://doi.org/10.5281/zenodo.7727577

Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 627–637.

Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. 2021. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*.

LLVM team. 2021. *SanitizerCoverage*. Retrieved April 8, 2022 from https://releases.llvm.org/11.0.1/tools/clang/docs/SanitizerCoverage.html

Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. 1949–1966.

Valentin JM Manès, Soomin Kim, and Sang Kil Cha. 2020. Ankou: Guiding grey-box fuzzing towards combinatorial difference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1024–1036.

Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* (2019).

Nathan Mantel et al. 1966. Evaluation of survival data and two new rank order statistics arising in its consideration. *Cancer Chemother Rep* 50, 3 (1966), 163–170.

Björn Mathis, Rahul Gopinath, and Andreas Zeller. 2020. Learning input tokens for effective fuzzing. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 27–37.

Stefan Nagy and Matthew Hicks. 2019. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 787–802.

Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. 2021. Same Coverage, Less Bloat: Accelerating Binary-only Fuzzing with Coverage-preserving Coverage-guided Tracing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 351–365.

Yannic Noller, Rody Kersten, and Corina S Păsăreanu. 2018. Badger: complexity analysis with fuzzing and symbolic execution. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 322–332.

Yannic Noller, Corina S Păsăreanu, Marcel Böhme, Youcheng Sun, Hoang Lam Nguyen, and Lars Grunske. 2020. HyDiff: Hybrid differential software analysis. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1273–1285.

Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 697–710.

Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 460–465.

Mohit Rajpal, William Blum, and Rishabh Singh. 2017. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596* (2017).

Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing.. In *Network and Distributed System Security*, Vol. 17. 1–14.

David A Schum. 2001. *The evidential foundations of probabilistic reasoning*. Northwestern University Press.

Dongdong She, Rahul Krishna, Lu Yan, Suman Jana, and Baishakhi Ray. 2020. MTFuzz: fuzzing with a multi-task neural network. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 737–749.

Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. Neuzz: Efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 803–817.

Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, Brent Byunghoon Kang, Jean-Pierre Seifert, and Michael Franz. 2020. Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In *29th USENIX Security Symposium (USENIX Security 20)*. 2541–2557.

Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution.. In *Network and Distributed System Security*, Vol. 16. 1–16.

Jonas Benedict Wagner. 2017. *Elastic program transformations: automatically optimizing the reliability/performance trade-off in systems software*. Ph. D. Dissertation. Ecole Polytechnique Fédérale de Lausanne.

Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 999–1010.

Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 579–594.

Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. 2018. SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 61–64.

Valentin Wüstholz and Maria Christakis. 2020. Targeted greybox fuzzing with static lookahead analysis. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 789–800.

Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2313–2328.

Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*. 745–761.

Michał Zalewski. 2014. *American fuzzy lop*. Retrieved April 8, 2022 from https://lcamtuf.coredump.cx/afl/

Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, and Yu Jiang. 2020. Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 858–870.

Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *29th USENIX Security Symposium (USENIX Security 20)*.

Received 2022-10-28; accepted 2023-02-25