DiPri: Distance-Based Seed Prioritization for Greybox Fuzzing (Registered Report)

Ruixiang Qian
qianrx@smail.nju.edu.cn
State Key Laboratory for Novel Software Technology
Nanjing University, China

Quanjun Zhang
quanjun.zhang@smail.nju.edu.cn
State Key Laboratory for Novel Software Technology
Nanjing University, China

Chunrong Fang*
fangchunrong@nju.edu.cn
State Key Laboratory for Novel Software Technology
Nanjing University, China

Zhenyu Chen
zychen@nju.edu.cn
State Key Laboratory for Novel Software Technology
Nanjing University, China

ABSTRACT

Greybox fuzzing is a powerful testing technique. Given a set of initial seeds, greybox fuzzing continuously generates new test inputs to execute the program under test and gravitates executions towards rarely explored program regions with code coverage as feedback. Seed prioritization is an important step of greybox fuzzing that prioritizes promising seeds for input generation. However, mainstream greybox fuzzers like AFL++ and Zest tend to slight the importance of seed prioritization and plainly pick seeds according to the order of the seeds being queued, or rely on an approach with randomness, which may consequently degrade their performance.

In this paper, we propose a novel distance-based seed prioritization approach named DiPri to facilitate greybox fuzzing. Specifically, DiPri calculates the distances among seeds and selects the ones that are farther from the others in priority to improve the probabilities of discovering previously unexplored regions. To make a preliminary evaluation, we integrate DiPri into AFL++ and Zest and conduct experiments on eight (four in C/C++ and four in Java) fuzz targets. We also consider six configurations, i.e., three prioritization modes multiplied by two distance measures, in our evaluation to investigate how different prioritization timings and measures affect DiPri. The experimental results show that, compared to the default seed prioritization approaches of AFL++ and Zest, DiPri covers 1.87%~13.86% more edges in three out of four C/C++ fuzz targets and 0.29%~4.97% more edges in the four Java fuzz targets with certain configurations. The results highlight the potential of facilitating greybox fuzzing with distance-based seed prioritization.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Greybox Fuzzing, Seed Prioritization, Seed Distance

ACM Reference Format:
Ruixiang Qian, Quanjun Zhang, Chunrong Fang, and Zhenyu Chen. 2023. DiPri: Distance-Based Seed Prioritization for Greybox Fuzzing (Registered Report). In *Proceedings of the 2nd International Fuzzing Workshop (FUZZING '23), July 17, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3605157.3605172

*Corresponding Author

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

ACM ISBN 979-8-4007-0247-1/23/07…$15.00
https://doi.org/10.1145/3605157.3605172

1 INTRODUCTION

Fuzzing is a software testing technique famous for its deployment simplicity and power in exposing potential software bugs [25]. The core of fuzzing is a loop where the fuzzer (the tool that implements fuzzing) continuously generates test inputs from a given set of initial seeds and executes the program under test (PUT) [27]. Greybox fuzzing is an essential branch of fuzzing that leverages execution status collected via lightweight instrumentation, e.g. code coverage, as the feedback for guidance [29]. Compared to blackbox fuzzing, which executes the PUT somewhat blindly, and whitebox fuzzing, which may incur huge costs due to adopting heavy program analysis techniques, greybox fuzzing achieves a better balance in effectiveness and efficiency [46]. The balance helps it discover crashes well and makes it a hot spot of recent year research [6, 23, 39, 42].

Seeds, or seed inputs, are critical for greybox fuzzing as they fundamentally shape the nature of the test inputs generated from them and thus can determine the trend of fuzzing to a large extent [9, 43]. In the workflow of greybox fuzzing, seed prioritization is performed to optimize promising seeds for subsequent input generation [36]. Although seed prioritization can greatly affect the performance of greybox fuzzing, its importance tends to be slighted by mainstream fuzzers. For example, the most famous C/C++ greybox fuzzer AFL [1] and the state-of-the-art Java greybox fuzzer Zest [32] both adopt an order-of-queued prioritization approach, which passes seeds to input generator by the order that they are added to the queue. In addition to the order-of-queued approach, AFL++ (a superior fork of AFL) provides a weighted random prioritization approach based on alias tables [2]. Both the above two approaches are somewhat blind and can starve the seeds that are actually promising for new coverage, which may consequently restrict fuzzers' performance.

Inspired by the thought of adaptive random testing (ART) that selects diverse tests according to the distances among tests [12, 18], we propose DiPri, a novel yet general distance-based seed

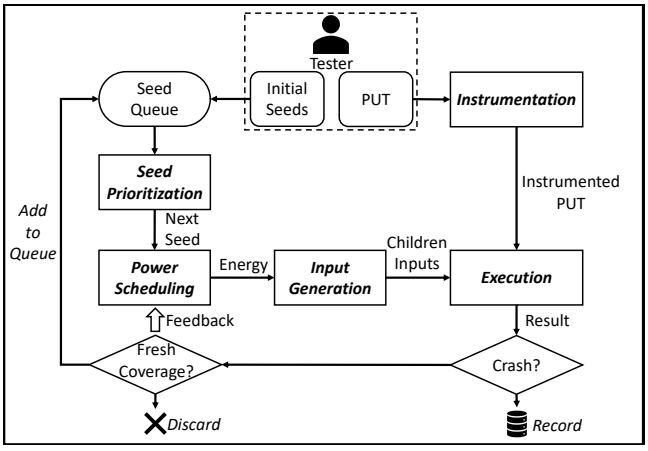


Figure 1: General workflow of greybox fuzzing

prioritization approach to boost greybox fuzzing. Once integrated into a greybox fuzzer, DiPri operates in two steps to rearrange seeds for input generation. First, DiPri prioritizes the queued seeds by reordering them according to the distances among them, where the distances are incrementally calculated at the times new seeds are added. Second, according to the orders, DiPri selects a seed for input generation from a certain range of the queue. Both the measures used for calculating seed distances and the ratio used for limiting the selection range can be customized by the testers.

We integrate DiPri into AFL++ and Zest and conduct experiments with eight fuzz targets (i.e., four for AFL++ and four for Zest) to make a preliminary evaluation. The prioritization approaches provided by AFL++ and Zest are chosen as baselines. Besides two distance measures, we come up with three prioritization modes to handle the possible overhead induced by distance calculation [13], which makes six DiPri configurations in total. The results involving 15 different fuzz campaigns over 1080 CPU hours show that, with certain configurations, DiPri can outperform AFL++'s default seed prioritization approach by 1.87%~13.86% in three targets except for `readelf` and outperforms Zest's default seed prioritization approach by 0.29%~4.97% in the four Java targets in terms of edge coverage. Through our experiments, we find that DiPri can incur non-negligible time costs during fuzzing, especially when calculating seed distances. To investigate the possibility of refining DiPri by accelerating distance calculations, we leverage edges per run, i.e., the number of edges covered via executing the PUT against a test input once, to evaluate the performance of DiPri from another perspective. Our analyses show that DiPri can cover more edges in the same times of runs compared to AFL++'s random-based prioritization approach in all four fuzz targets, which implies that DiPri can be further improved with a more efficient distance calculation.

2 BACKGROUND AND MOTIVATION

2.1 Greybox Fuzzing

Greybox fuzzing is an essential family of fuzzing [25, 46]. Figure 1 illustrates the general workflow of greybox fuzzing. To make a preparation, greybox fuzzing first constructs the seed queue with the initial seeds provided by the tester and instruments the PUT for tracing code coverage at run-time. It then operates in a loop, i.e., the main fuzz loop, to continuously generate new test inputs to execute the PUT, which includes four main steps: 1) ***Seed Prioritization***, 2) ***Power Scheduling***, 3) ***Input Generation***, and 4) ***Execution***. At first, greybox fuzzing prioritizes seeds and selects one for input generation. It next conducts power scheduling to assign energy, i.e., time budget for input generation, to the selected seed [6]. Greybox fuzzing then repeatedly generates children inputs by imposing various mutators upon the parent seed until the energy runs out [14, 37] Every child input is executed against the PUT to explore code regions and probe unknown crashes. According to the execution result, a child input can be 1) recorded if it has triggered any crash, or 2) added to the queue if has achieved fresh coverage.

Seed prioritization is a critical step of greybox fuzzing [36] that determines the trends of fuzzing by rearranging the queued seeds and optimizing the most promising ones for subsequent rounds of input generation. Existing greybox fuzzing techniques tend to slight seed prioritization. In this paper, we propose a distance-based seed prioritization approach, named DiPri, to facilitate greybox fuzzing. The designs of DiPri are described in Section 3.

2.2 Distance-based Test Selection

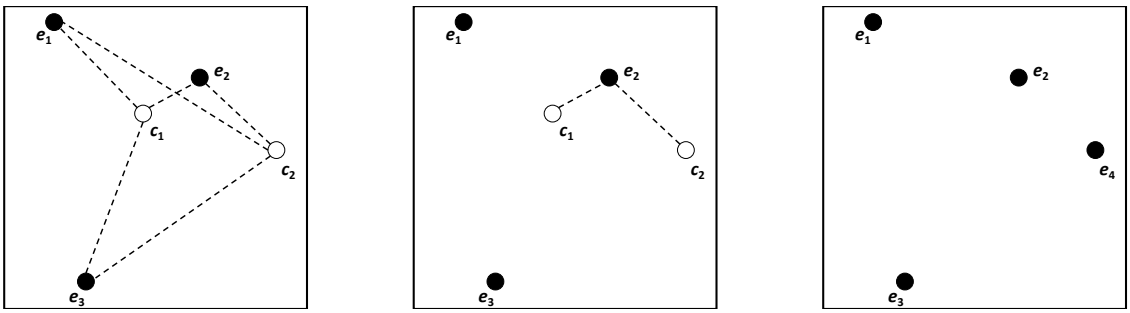
The essential insight behind adaptive random testing (ART) is to select tests from different code regions, as failures tend to cluster into contiguous regions [12, 18, 35]. With various intuitions, ART can be implemented in many different ways [18]. Fixed-Size-Candidate-Set (FSCS) is one of the most widely adopted implementations of ART, which picks a fixed-size of tests at each iteration of selection [12, 18]. Suppose C is the set of candidate tests for selection and E is the set of tests that have already been executed. The target of FSCS is to pick k ($k \in N^*$) tests that are farthest away from E with respect to a certain constraint. For example, a common constraint is to select a test that has the maximum minimum distance to all the executed tests, which can be denoted as follows:

$$\tau: \ \forall c \in C, \ \min_{e \in E} \mathcal{D}(c', e) \geq \min_{e \in E} \mathcal{D}(c, e) \tag{1}$$

where τ is the notation of the constraint, $\mathcal{D}$ is the distance function for calculating the dissimilarity between two tests, c and e are the elements of C and E, an c' is the candidate test selected in an iteration of FSCS satisfying τ. Figure 2 illustrates the process of FSCS complying the constraint τ, where the executed test set is $E = \{e_1, e_2, e_3\}$ and the candidate set is $C = \{c_1, c_2\}$. To select a test from C for the next execution, FSCS first calculates the distance $\mathcal{D}(e, c)$ between each pair of candidate and executed test (Figure 2a). The distances between tests are represented with dashed lines. With the calculated distances, FSCS identifies the candidate test that is the farthest from the executed ones with considered constraint τ (Figure 2b). The candidate test c_2 is selected as the next test e_4 for execution as it has a larger minimum distance than c_1 (Figure 2c).

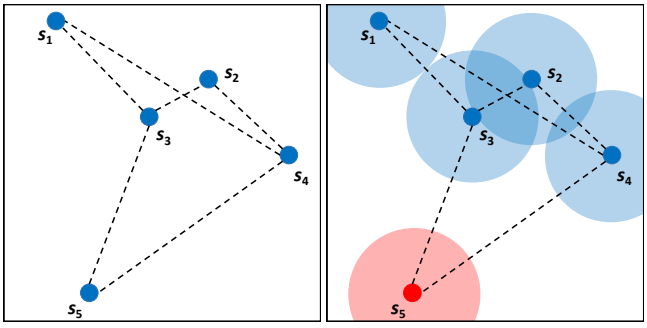
2.3 A Motivating Example

We present our targeted problem scope and show how DiPri works through a simple example. Given a queue of seeds S, the target of DiPri is to prioritize and select an outlier seed, which is the farthest from the others, for input generation. Figure 3 illustrates a



(a) Calculating test distances. (b) Comparing minimum distances. (c) Selecting next candidate.

Figure 2: The exemplified process of FSCS complying the constraint τ.



(a) Calculating seed distances. (b) Selecting the outlier seed s_5.

Figure 3: An illustration of selecting an outlier seed.

simplified scenario that DiPri may face during fuzzing, where the seed queue $S = \{s_1, s_2, s_3, s_4, s_5\}$. The subscripts ($1 \sim 5$) represent the order in which the seeds are added to the queue. The light-colored circles in Figure 3b simulate the regions that can be explored by the test inputs generated from the seeds at the circle centers. Note that we ignore the possibility that the selected seeds can give birth to test inputs that are able to explore very remote regions for simplicity [41]. Recall the two prioritization approaches adopted by the mainstream fuzzers (i.e., AFL, AFL++, and Zest) described in Section 1, as both unaware of the distribution of seeds, the order-of-queued approach trivially picks a seed for input generation in order (from s_1 to s_5) and thus drives fuzzer to explore overlapped regions repeatedly; meanwhile, the random approach is more likely to pick seeds from $\{s_1, s_2, s_3, s_4\}$, which may also drive the fuzzers to visit densely explored regions. Unlike these two approaches, DiPri calculates the distances among seeds to identify their distributions (Figure 3a) and prioritize the outlier seed s_5 (colored in red in Figure 3b) to choose next. As such, greybox fuzzing can avoid densely explored regions and save resources for rarely explored ones.

3 APPROACH

We elaborate on the designs of DiPri in this section. Algorithm 1 illustrates the greybox fuzzing equipped with DiPri, where the additional parts are colored in blue. Once the seed queues are initialized (line 1) and the main fuzz loop starts (line 2), DiPri prioritizes

Algorithm 1 Greybox Fuzzing with DiPri

Input: Initial Seeds S, Instrumented PUT p, DiPri Config *conf*
Output: Seed Queue S', Crashing Seed Queue S^*

```
1:  S' ← S, S* ← ∅
2:  repeat                                          ▷ Main fuzz loop
3:      S' ← prioritizeByDistance(S', conf)
4:      s ← selectNext(S', conf)
5:      e ← assignEnergy(s)
6:      for i from 1 to e do
7:          s' ← generateNewInput(s)
8:          res ← execute(p, s')
9:          if isCrash(s', res) then
10:             add s' to S*
11:         else if isInteresting(s', res) then
12:             vec ← parseDistVec(res, conf)
13:             bindInputWithVec(s', vec)
14:             add s' to S'
15:         end if
16:     end for
17: until resources exceeds or abort-signal comes
18: return S', S*
```

the seed queue with respect to the configuration *conf* given by the tester at the beginning of each iteration (line 3). The details of the distance-based seed prioritization are described in Section 3.2. After prioritizing seeds, a seed is picked from the queue S' and assigned energy to generate children inputs (line 4~16). During input generation, every child input is executed against the instrumented PUT to produce run-time behaviors (line 7~8). The inputs that lead to crashes are added to the crashing seed queue S^* for later reproduction (line 10), and the interesting seeds that attain fresh coverage are added to S' to support subsequent fuzzing (line 14). Before adding the interesting input s' to S', DiPri first parses the feature vector *vec* (e.g., coverage vector) according to *conf* and then binds it to s' (line 12~13). The vector *vec* is later used for calculating seed distances in the next prioritization (Section 3.1).

3.1 Calculating Seed Distances

Different distance measures can quantify seed distances from various perspectives. For example, the Euclidean distance can measure

the dissimilarities among numeric inputs [18, 21], whereas the Jaccard distance can measure the dissimilarities among the coverage vectors achieved by tests [33]. In this paper, we focus on the coverage distances and leverage coverage as a feature to quantify seed distances. Besides Jaccard distance [19], we also consider Hamming distance [30] in our evaluation. The details of the distance measures are presented in Section 4.1. Suppose S is a seed queue consisting of n ($n \in N^*$) seeds and s_i, s_j ($i, j \in 1, 2, ..., n$) are two seeds from S. Let vec_i, vec_j be the coverage vectors of s_i, s_j and $\mathcal{D}$ be the distance function. The distance between s_i and s_j is defined as the distance between their vectors, which satisfies:

$$\mathcal{D}(s_i, s_j) = \mathcal{D}(vec_i, vec_j) \tag{2}$$

To identify and prioritize the outlier seed that is the farthest from the others, we define the seed distance as the sum of the distances between a seed s_i and the other seeds within S. With Equation 2, the calculation for the seed distance of s_i can be formalized as:

$$\mathcal{D}(s_i) = \sum_{i=j}^{n} \mathcal{D}(s_i, s_j) = \sum_{i=j}^{n} \mathcal{D}(vec_i, vec_j) \tag{3}$$

3.2 Prioritizing Seeds by Distance

Algorithm 2 Distance-based Seed Prioritization

Input: Seed Queue S, DiPri Config *conf*
Output: Prioritized Seed Queue S_p

```
1:  S_p ← S
2:  n ← getQueueLen(S)
3:  if hasGrown(S) and reachTimming(conf) then
4:      for i from 1 to n do                          ▷ Update seed distances.
5:          s_i ← getSeed(S_p, i)
6:          if isFresh(s_i) then                      ▷ Focus on freshly added seeds.
7:              for j from 1 to n do
8:                  if i ≠ j then
9:                      s_j ← getSeed(S_p, j)
10:                     d ← calDist(s_i, s_j, conf)
11:                     incSeedDist(s_i, d)
12:                     incSeedDist(s_j, d)
13:                 end if
14:             end for
15:         end if
16:     end for
17:     S_p ← sortByDist(S_p)                         ▷ Prioritize.
18: end if
19: return S_p
```

DiPri prioritizes seeds at the beginning of each iteration of the main fuzz loop. With the notations introduced in Section 3.1, the constraint τ' that DiPri complies for prioritizing is formalized as:

$$\tau' : \forall s_i, s_j \in S,\ \mathcal{D}(s_i) \geq \mathcal{D}(s_j) \implies s_i \succeq s_j \tag{4}$$

where $s_i \succeq s_j$ denotes that s_i is prioritized before or equally to s_j. Algorithm 2 illustrates the process of the seed prioritization with τ'. First of all, DiPri checks whether it is time to perform prioritization (line 3). Specifically, the prioritization condition is decided by two factors: 1) whether the queue S has grown in previous fuzzing and 2) whether the prioritization timing is reached with respect to the prioritization mode given in *conf*. The prioritization timings vary for different modes. For example, under the `VANILLA` mode, DiPri prioritizes the seed queue S every time the queue has grown; whereas under the `PERIODICAL` mode, DiPri prioritizes S when the time elapsed since the last prioritization has exceeded the preset period. We elaborate on the configurations of prioritization modes in Section 4.2. Next, DiPri updates seed distances for queued seeds (line 4∼16). To reduce the overhead brought by calculating distances, we only focus on seeds that are newly added since the last prioritization (line 6) and increase maintained seed distances with the newly calculated ones (line 11∼12). After updating seed distances, DiPri further prioritizes S_p by adjusting the order of seeds within the queue (line 17). The prioritized seed queue S_p is finally returned to assist subsequent input generation (line 19).

3.3 Picking Next Seeds in Range

Algorithm 3 Seed Selection in Range

Input: Prioritized Seed Queue S_p, DiPri Config *conf*
Output: Picked Seed s

```
1: n ← getQueueLen(S_p)
2: r ← getSelectRatio(conf)
3: u ← n * r                          ▷ Calculate the upper range of selection
4: i ← getLastSeedIdx(S_p)
5: i ← (i + 1)%u                      ▷ Move seed pointer to the next
6: s ← getSeed(S_p, i)
7: return s
```

Concisely, DiPri picks the next seed for input generation according to the order of seeds after prioritization. Algorithm 3 illustrates the process of picking the next seed. The tester can adjust the selection ratio r within *conf* to configure the range of picking (line 2∼3). The selection ratio r is a float value range in $(0, 1]$ that implies the testers' preference of top-u seeds. We set r to 1 by default in current experiments, which means all the seeds in S_p are electable. After determining the range, DiPri gets the seed index i that now points to the last selected seed (line 4) and advances i to the next position with respect to the upper range u (line 5). The picked seed is obtained by the advanced index i (line 6) and finally returned.

4 EVALUATION

We conduct several experiments to preliminarily evaluate DiPri. In this section, we first elaborate on the configuration items we consider (Section 4.1∼4.2) in our evaluation, and then describe the experimental setups (Section 4.3). After that, we present the experimental results and analyze them to mine insights (Section 4.4∼4.5). We focus on the following research questions in the evaluation:

- **RQ1:** Can DiPri facilitate greybox fuzzing?
- **RQ2:** How much time is consumed by DiPri?

4.1 Distance Measures

We include two distance measures, i.e., Hamming distance and Jaccard distance in our evaluation. We first describe the definition of coverage vector and then describe the calculations of distances.

4.1.1 Coverage Vector. Both Hamming and Jaccard distances as binary distances that can be leveraged to quantify the distance between two sequences of binary values [19, 30]. We define the elements of the coverage vectors as the flags of whether the program entities are covered, which are binary values ranging in $\{0, 1\}$. Let s_a be a seed and k be the total number of the entities of the PUT ($k, a \in N^* \wedge a \leq k$). Let f_a^i ($f_a^i \in \{0, 1\}$) be the binary value representing whether the program entity e_i is covered by s_a where 1 implies e_i is covered and 0 implies missed. The coverage vector vec_a of s_a can be defined as:

$$vec_a = \{f_a^1, f_a^2, ..., f_a^k\} \tag{5}$$

4.1.2 Hamming and Jaccard Distances. The Hamming distance is defined as the number of different elements between two sequences of binary codes [30], while the Jaccard distance is defined as the supplementary ratio between the intersection and union of two sets [19]. Both Hamming and Jaccard distances are adopted in prior research for measuring test dissimilarities [33, 38]. According to the definitions, with Equation 2 and Equation 5, the Hamming distance $\mathcal{D}_H$ and Jaccard distance $\mathcal{D}_J$ between s_a and s_b are calculated as:

$$\mathcal{D}_H(s_a, s_b) = \mathcal{D}_H(vec_a, vec_b) = \sum_{i=1}^{k}(f_a^i \oplus f_b^i)) \tag{6}$$

$$\mathcal{D}_J(s_a, s_b) = \mathcal{D}_J(vec_a, vec_b) = 1 - \frac{\sum_{i=1}^{k}(f_a^i \wedge f_b^i)}{\sum_{i=1}^{k}(f_a^i \vee f_b^i)} \tag{7}$$

4.2 Prioritization Modes

We provide three types of prioritization modes in our evaluation, i.e., the `VANILLA` mode, the `ADAPTIVE` mode, and the `PERIODICAL` mode. `VANILLA` is the most trivial mode that prioritizes the seed queue every time fresh seeds are added. The frequent prioritization may incur too much extra overhead and thus slow down the speed of fuzzing. Therefore, hoping to improve the efficiency of DiPri, we further design `ADAPTIVE` and `PERIODICAL` modes on top of the `VANILLA`. Other than checking whether the seed queue has grown, the `ADAPTIVE` mode prioritizes the seed queue at the time that all last prioritized seeds have already been exhausted for input generation; whereas the `PERIODICAL` prioritizes the seed queue once the time elapsing since the previous prioritization has exceeded the period δ that is set the user in advance. Note that the `PERIODICAL` mode also prioritizes the seed queue when all the last prioritized seeds are exhausted, which means that the `PERIODICAL` mode can convert to the `ADAPTIVE` mode if the preset period is too long.

4.3 Experimental Setups

Our experimental setups consist of seven parts, i.e., the fuzzers we integrated with DiPri, the implementation of DiPri on different host fuzzers, the fuzz targets, the baseline seed prioritization approaches, DiPri configurations, fuzz campaigns settings, and the infrastructures. We next elaborate on these setups in order.

4.3.1 Fuzzers. We include AFL++ [14] and Zest [32] in our evaluation. AFL++ is a superior fork of the well-known greybox fuzzer AFL [1]. It is community-driven and has integrated with many efforts from both academia and industry, such as the mutation operator scheduling of MOpt [24], the input-to-state correspondence optimization of Redqueen [3], and exploration strategy of AFLFast [7]. Zest is a generator-based fuzzer implemented on the Java fuzzing platform JQF [31]. We choose AFL++ and Zest as they are widely adopted in fuzzing research [8, 20, 29, 45]

4.3.2 Implementations. We implement DiPri in both C and Java and integrate it into AFL++ (version 4.06c) and Zest (version 2.0), respectively. The integrations that cross fuzzers and programming languages reveal the good scalability of DiPri. For AFL++, we mainly modify the `afl-fuzz-queue.c` file to record coverage vectors for seeds before adding them to the queue, and provide functions for distance calculation and seed prioritization. For Zest, we construct a new class hierarchy for distance calculation and extend `ZestGuidance` to support DiPri. We also modify the maven MOJO of JQF to supply arguments for different DiPri configurations.

Table 1: Details of fuzz targets

	Project	Version	Target	Input Type
C/C++	Binutils	2.40	**`cxxfilt`**	String
			`objdump`	ELF
			`readelf`	ELF
	Libxml2	2.9.11	**`xmllint`**	XML
Java	Ant	1.10.2	**`ant`**	XML
	Bcel	6.2	**`bcel`**	Java Bytecode
	Closure	20180204	**`closure`**	JS
	Rhino	1.7.8	**`rhino`**	JS

4.3.3 Fuzz Targets. We include eight fuzz targets, i.e., four C/C++ targets and four Java targets, in our evaluation. The details of the fuzz targets are shown in Table 1, which displays the project names, project versions, fuzz target names, and input types. For AFL++, the fuzz targets are executable binaries that are instrumented with code for monitoring coverage. We build fuzz targets for AFL++ by leveraging `afl-cc` to instrument the projects under the default setting. For Zest, we adopt the four fuzz targets previously used in its evaluation to make a fair comparison [32]. Zest is implemented upon the JQF platform [31], where the fuzz targets are Junit-style test methods annotated with `@Fuzz`. For example, the four Java fuzz targets shown in Table 1 are test methods named `testWithGenerator` implemented in different test classes. For distinction, we use the lowercase of the project names, i.e., **`ant`**, **`bcel`**, **`closure`**, and **`rhino`** to denote the four Java targets. As for initial seeds, we provide "`_Z1fv`" for **`cxxfilt`** and AFL++ test cases for the other three C/C++ targets following the setup of FairFuzz [22], and provide no seeds for the four Java targets following the setup of Zest [32]. Note that Zest does not strictly require initial seeds as it relies on generators to synthesize inputs of certain types [31].

4.3.4 Baselines & DiPri Configurations. We take three seed prioritization approaches, i.e., two from AFL++ and one from Zest, as baselines in our evaluation. By default, AFL++ adopts a weighted random prioritization based on an alias table [2]. When specified with `-Z` option in the command line, AFL++ turns to use the old order-of-queued approach previously adopted by AFL [1]. For clarity, we mark these two prioritization approaches as `AFLPP` and

Table 2: The total number of covered edges (ΣEdges) and the average EPR values (μEPR) that are achieved by different seed prioritization approaches over the three repetitions. The red down arrows alongside percentages represent decreases to the fuzzers' default seed prioritization approach, i.e., AFLPP or ZEST, while the green up arrows represent increases.

Trgt. / Appr.	cxxfilt		objdump		readelf		xmllint	
	ΣEdges	μEPR	ΣEdges	μEPR	ΣEdges	μEPR	ΣEdges	μEPR
AFLPP	7332 —	2.0E-05 —	11050 —	4.6E-05 —	15454 —	4.8E-05 —	8076 —	3.9E-05 —
AFLPP-Z	7150 ↓2.48%	2.6E-05 ↑27.96%	11218 ↑1.52%	4.9E-05 ↑7.94%	14804 ↓4.21%	4.9E-05 ↑2.98%	8628 ↑6.84%	5.1E-05 ↑31.88%
DIST-VH	7235 ↓1.32%	2.6E-05 ↑25.60%	11054 ↑0.04%	6.5E-05 ↑41.84%	15190 ↓1.71%	8.4E-05 ↑76.01%	8277 ↑2.49%	4.8E-05 ↑23.56%
DIST-VJ	7031 ↓4.11%	2.5E-05 ↑24.05%	11469 ↑3.79%	5.7E-05 ↑24.10%	14956 ↓3.22%	8.7E-05 ↑82.88%	8642 ↑7.01%	4.8E-05 ↑24.05%
DIST-AH	7469 ↑1.87%	2.8E-05 ↑39.32%	11083 ↑0.30%	5.4E-05 ↑17.71%	15095 ↓2.32%	7.0E-05 ↑45.90%	8946 ↑10.77%	5.2E-05 ↑34.76%
DIST-AJ	7375 ↑0.59%	2.6E-05 ↑27.23%	10367 ↓6.18%	5.4E-05 ↑18.93%	14511 ↓6.10%	8.7E-05 ↑81.87%	9195 ↑13.86%	5.5E-05 ↑42.89%
DIST-PH	7346 ↑0.19%	3.1E-05 ↑52.93%	10899 ↓1.37%	5.6E-05 ↑23.80%	14968 ↓3.14%	6.9E-05 ↑44.63%	8331 ↑3.16%	4.8E-05 ↑24.22%
DIST-PJ	7286 ↓0.63%	3.0E-05 ↑49.87%	11465 ↑3.76%	5.5E-05 ↑20.70%	15105 ↓2.26%	8.7E-05 ↑82.62%	8316 ↑2.97%	4.7E-05 ↑20.40%
Java								
Trgt. / Appr.	ant		bcel		closure		rhino	
	ΣEdges	μEPR	ΣEdges	μEPR	ΣEdges	μEPR	ΣEdges	μEPR
ZEST	5482 —	7.1E-04 —	7935 —	1.5E-04 —	47044 —	7.2E-03 —	17604 —	1.9E-03 —
DIST-VH	4649 ↓15.20%	6.8E-05 ↓90.42%	7414 ↓6.57%	4.1E-04 ↑171.62%	44999 ↓4.35%	3.8E-03 ↓47.52%	16911 ↓3.94%	3.9E-04 ↓79.69%
DIST-VJ	4886 ↓10.87%	7.6E-05 ↓89.28%	7360 ↓7.25%	6.7E-05 ↓55.09%	44368 ↓5.69%	2.1E-03 ↓70.83%	16773 ↓4.72%	4.6E-04 ↓76.06%
DIST-AH	5495 ↑0.24%	6.2E-04 ↓13.36%	8018 ↑1.05%	1.4E-04 ↓9.35%	47182 ↑0.29%	7.8E-03 ↑9.21%	17685 ↑0.46%	1.6E-03 ↓14.99%
DIST-AJ	5493 ↑0.20%	7.0E-04 ↓1.21%	7943 ↑0.10%	4.3E-04 ↑187.97%	46824 ↓0.47%	6.1E-03 ↓15.05%	17435 ↓0.96%	1.5E-03 ↓19.32%
DIST-PH	5488 ↑0.11%	6.4E-04 ↓9.70%	8030 ↑1.20%	1.9E-04 ↑25.47%	44926 ↓4.50%	8.5E-03 ↑18.22%	17137 ↓2.65%	2.8E-03 ↑46.69%
DIST-PJ	5518 ↑0.66%	7.1E-04 ↓1.07%	7880 ↓0.69%	5.5E-04 ↑268.59%	44922 ↓4.51%	3.3E-03 ↓53.91%	18479 ↑4.97%	6.3E-04 ↓66.94%

AFLPP-Z. Zest [32] only provides one default seed prioritization that picks the next seeds according to the order of the seeds within the queue, which is similar to AFLPP-Z. We mark Zest's default seed prioritization approach as ZEST. As for DiPri, we consider two distance measures, i.e., Hamming and Jaccard distances, and three modes, i.e., VANILLA, ADAPTIVE, and PERIODICAL in our evaluation, which makes six configurations in total. To mark DiPri configurations, we connect the prefix DIST with the initials of modes (V, A, or P) and distance measures (H or J). In particular, the six DiPri configurations are DIST-VH, DIST-VJ, DIST-AH, DIST-AJ, DIST-PH, and DIST-PJ. We set the selection ratio r (Section 3.3) to 1 (the default value) in all the experiments and the prioritization period δ (Section 4.2) to 60 seconds for all configurations involving PERIODICAL.

4.3.5 Fuzz Campaigns & Infrastructures. A Fuzz campaign is a specific execution of a fuzzer on a specific fuzz target lasting for a period of time [25]. We follow prior works to set the duration of campaigns to 24 hours [29, 44] and treat a fuzzer (i.e. AFL++ or Zest) that is equipped with a certain seed prioritization approach as a specific fuzzer. We repeat each fuzz campaign for three times to alleviate the influences of randomness. In total, our experiments involve 15 fuzz campaigns (eight with AFL++ and seven with Zest) and take 1080 CPU hours. All the experiments run on a machine with 16 cores (Intel® Xeon® Silver 4114 CPU @ 2.20GHz) and 32 GB memory running Ubuntu 18.04 operating system.

4.4 RQ1: Effectiveness of DiPri

4.4.1 Metrics. We consider two effectiveness metrics in our preliminary experiment. The first metric is code coverage. Both AFL++ and Zest provide statistical components to record the number of covered edges, i.e., **#Edges**, during fuzzing. Slightly different from AFL++, Zest focuses on generating semantic-valid test inputs and additionally records the number of edges covered by valid inputs, i.e., **#Valid Edges**. Therefore, we adopt **#Edges** as the coverage metric for AFL++ and **#Valid Edges** for Zest to follow its research focus. The second metric is executions per run (**EPR**), which essentially shows how effective different seed prioritization approaches are in covering edges in the same number of runs against different test inputs. As distance calculation can incur extra overhead, we introduce **EPR** to reveal whether it is possible to further refine DiPri by accelerating the calculation. Formally, **EPR** is calculated as the ratio of the number of (valid) covered edges and the number of total runs, which can be denoted as $\textbf{EPR} = \frac{\textbf{\#Edges}}{\textbf{\#Runs}}$. The number of runs is represented differently by different fuzzers. Specifically, in our evaluation, we adopt the number of executions recorded by AFL++ and the number of trials recorded by Zest as **#Runs**.

4.4.2 Effectiveness on Covered Edges. The **ΣEdges** columns of Table 2 show the total edges covered by the fuzzers equipping different seed prioritization approaches in three repeated 24-hour fuzz campaigns. The baseline prioritization approaches, i.e., AFLPP, AFLPP-Z and ZEST are backed in grey and the DIST-family is in white. The red down arrows show the decreases compared to fuzzers' default seed prioritization approaches, i.e., AFLPP and ZEST, and the green up arrows show the increases. The decreased/increased percentages are tagged alongside the arrows. Considering all six configurations, DiPri outperforms AFLPP in three out of four C/C++ targets and ZEST in all four Java targets in terms of **ΣEdges**. In total, DiPri achieve the best in seven out of eight fuzz targets except for **readelf**. We further compare the trends of covering edges during fuzzing. To this end, we depict curves with averaged number of covered edges as the dependent variable and the fuzzing time as

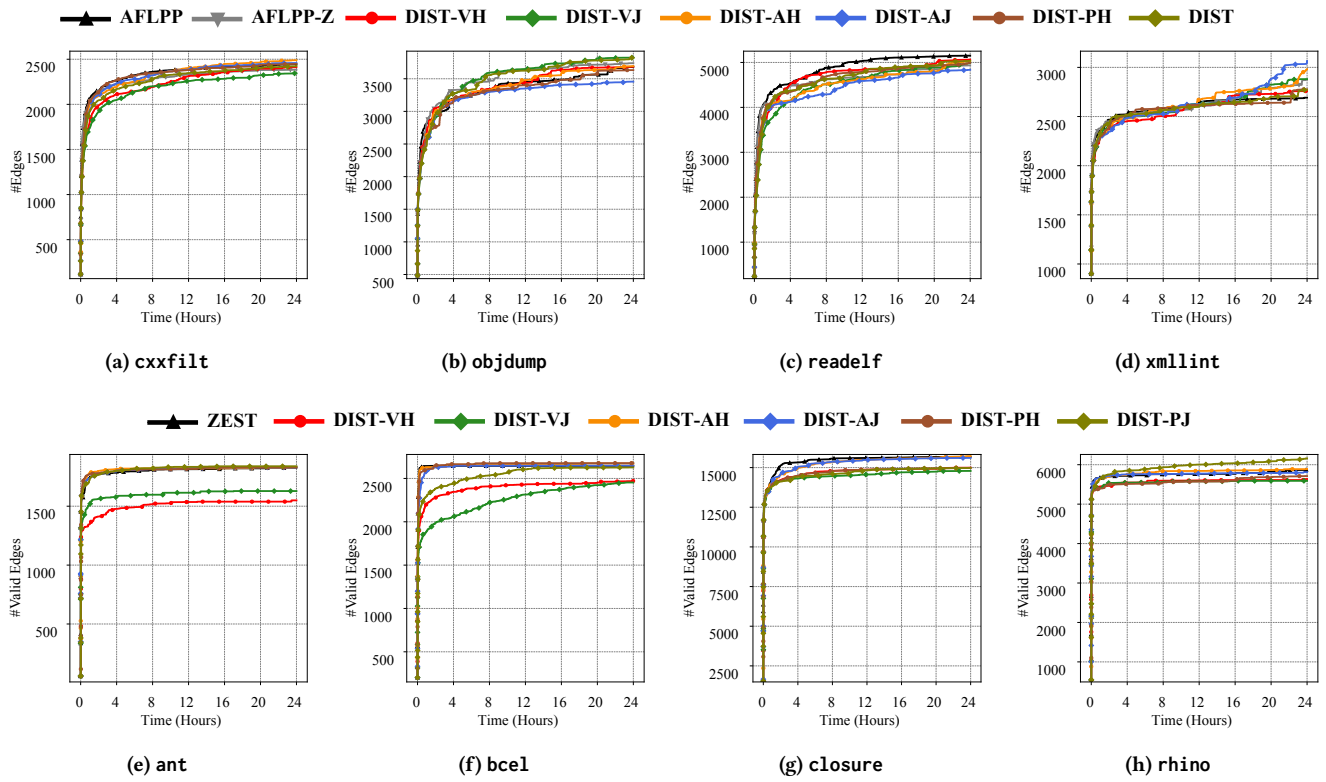


Figure 4: Edges covered by different seed prioritization approaches in 24 hours averaged on three repetitions. The Y-axis is the number of covered (valid) edges while the X-axis is the elapsed time. The legends are tagged upon the subfigures.

the independent variable. Figure 4 displays the curves of covering edges achieve by the fuzzers that equip different seed prioritization approaches in 24 hours, where Figure 4a~4d are curves of C/C++ targets and Figure 4e~4h are of Java targets. The DIST-family outperforms the baseline approaches (i.e., AFLPP, AFLPP-Z and ZEST) in six out of eight targets, except for **readelf** in C/C++ and **closure** in Java. Considering the extra overhead incurred by distance calculation, it is quite surprising that DiPri can outperform the baseline seed prioritization approaches in most of the fuzz targets in terms of not only Σ**Edges** but also the efficiency of covering edges.

4.4.3 Effectiveness on EPR. We calculate the averaged **EPR** values for each seed prioritization approach to unveil their capability of helping the fuzzers to cover edges in the same times of runs. The μ**EPR** columns of Table 2 show the mean **EPR** values achieved by different seed prioritization approaches averaged over three repetitions. As for C/C++ targets, the six DiPri outperform AFLPP in all the four targets. The increased percentages range from 17.71% (by DIST-AH in **objdump**) to 82.88% (by DIST-VJ in **readelf**) and achieve an average increase percentage of 39.58%. The results on C/C++ targets imply that the performance of DiPri can be further improved by accelerating distance calculation. However, as for Java targets, things are a bit different. Although the DIST-family outperforms ZEST in three out of four fuzz targets except for **ant** and achieves an average increase percentage of 0.58%, the results are volatile for different configurations. When picking the V prioritization mode (i.e., DIST-VH and DIST-VJ), DiPri can even deteriorate the performance of fuzzers some Java targets in terms of **EPR**. To explain this situation, we record and analyze the seed picked by DiPri at each fuzz iteration, and find that our DiPri implementation on Zest tends to pick the few identical seeds added at the early stage of fuzzing. These seeds are usually small and fast for execution, which can increase the number of runs during the same fuzzing duration and thus can consequently decrease **EPR** values.

4.5 RQ2: Time Overhead of DiPri

We record the time consumed by DiPri during fuzzing in our C/C++ implementation to investigate how much time overhead is brought by DiPri. Figure 5 displays the percentages of time occupied by DiPri (colored in blue) and other original fuzzing components (colored in red) in 24-hour fuzz campaigns repeating for three times. The results are grouped by distance measures or prioritization modes (H~P), DiPri configurations (VH~VJ), and the fuzz targets (T1~T4). We leverage T1, T2, T3, T4 to respectively stand for C/C++ fuzz targets **cxxfilt**, **objdump**, **readelf**, **xmllint** for clarity. With Figure 5, one can observe that the ratios of the time consumed by DiPri vary across different configurations and fuzz targets. The occupation percentages are generally below 20.0% except for **readelf** (T3). Comparing the two considered distance measures, i.e., Hamming (H) and Jaccard (J), the time consumed by calculating Jaccard

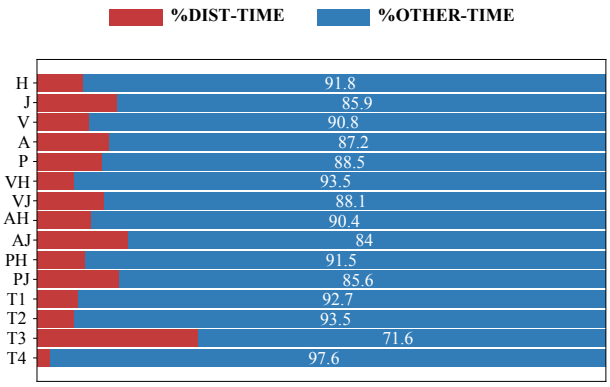


Figure 5: The percentage of the time consumed by DiPri (red) and other components (blue) during fuzzing.

distances is a bit more (+5.9%) than calculating Hamming distances. This result is attributed to the different implementations of calculating Hamming and Jaccard distances, where one binary computation, i.e., $f_i \oplus f_j$, is operated for each coverage flag f_i when calculating Hamming, while two binary computations, i.e., $f_i \wedge f_j$ and $f_i \vee f_j$, are operated when calculating Jaccard (Section 4.1). Comparing the three prioritization modes, i.e., `VANILLA` (`V`), `ADAPTIVE` (`A`) and `PERIODICAL` (`P`), the time consumed by `VANILLA` is the least (9.2%), where the time consumed by `ADAPTIVE` is the most (12.8%). This result is out of our expectations as we used to estimate that `VANILLA`, which drives its host fuzzers to execute the distance-based seed prioritization process every time the seed queues are updated, would cost the most time. The time differences in the two distance measures and three prioritization modes lead to the time differences in the six configurations, where the configuration `DIST-AJ` cost the most time (16.0%) and `DIST-VH` cost the least (6.5%). As for the four fuzz targets, DiPri consumes less than 10.0% of time budget in **`cxxfilt`** (`T1`), **`objdump`** (`T2`) and **`xmllint`** (`T4`), while consumes 28.4% of time in **`readelf`** (`T3`). This result explains the decreases of **#Edge** and the increases of **EPR** achieved by the `DIST`-family in **`readelf`** (Table 2) and implies that the overhead of distance calculation is also influenced by the nature of fuzz targets.

4.6 Further Evaluation

Although our preliminary experiments have revealed the potential of DiPri to some extent, there is still a lot to investigate. In particular, we summarize the further evaluation as follows:

- **More configurations.** DiPri provides configuration items for customizing, including the types of features used for calculating seed distances (distance feature), the distance measures, the prioritization modes, and the selection ratio. In spite of code coverage, seeds' content (e.g., values of certain program variables) is an essential feature that can largely influence the performance of greybox fuzzing [4, 20]. Besides Jaccard and Hamming, Euclidean distance is another measure widely adopted for quantifying the dissimilarities between the content or coverage of tests.
- **More experiments.** In our preliminary evaluation, we merely conduct experiments on eight fuzz targets, repeat each fuzz campaign for three times, and leverage code coverage to indicate the effectiveness of DiPri. Future works will involve more varied fuzz targets, more repetitions, and more analyses including 1) the differences between the code regions that can be explored with or without DiPri equipped, 2) the impacts that DiPri can cause to bug-finding capabilities, 3) the nature of the seeds that are prone to be selected by DiPri, 4) the essential pros and cons of the proposed prioritization modes, 5) the best choice of the selection ratio r (Section 3.3), and 6) the individual contributions of the seed prioritization and the distance metrics. Besides, the comparison study to other similar works is worth taking.

5 RELATED WORK

Greybox Fuzzing. Many works focus on improving greybox fuzzing by upgrading its components [17, 23, 39, 42], such as seed scheduling [36, 40], input generation [24, 43], execution guidance [22, 23], and power scheduling [5, 44]. Seeds are critical for greybox fuzzing as they shape the input generated at run-time. Therefore, some fields like seed selection and seed prioritization dedicate to ensuring the quality of seeds either ahead of fuzzing starts or input generation. In particular, seed selection aims at preparing high-quality seed corpora for greybox fuzzing. For example, Rebert et al. [34] crawl seeds from the web and introduce heuristics to refine them for formal fuzzing; Herrera et al. [15] conduct large-scale experiments to investigate how initial seeds impact the performance of greybox fuzzers and propose a constraint-based corpus minimization tool named OptiMin. In contrast to their works, we focus on the quality of the seeds used for input generation and propose DiPri to prioritize outlier seeds according to seed distances. As for seed prioritization, She et al. [36] propose a seed scheduler named K-Scheduler to identify, prioritize and optimize seeds at the horizons of explored and unexplored code regions. Unlike K-Scheduler, DiPri neither requires centrality analysis to identify horizon seeds nor focuses on optimizing them via allocating extra energy.

Adaptive Random Testing. ART is a long-standing research topic that has been studied for decades [10–12, 16]. The main application scenario of ART is the unit testing of toy programs requiring numeric test inputs [10, 26]. In this paper, we depart from the key insight behind ART (i.e., selecting diverse tests from different regions) and propose a distance-based seed prioritization approach for greybox fuzzing. In contrast to the classic application scenario of ART, greybox fuzzing is typically considered a system testing technique that targets complex real-world projects [28]. Our work is quite different from previous ART research in this respect.

6 CONCLUSION AND FUTURE WORK

In this paper, we propose DiPri to prioritize seeds by distances for greybox fuzzing. We implement DiPri on top of two mainstream greybox fuzzers, i.e., AFL++ and Zest, and conduct experiments on eight fuzz targets to preliminarily evaluate it. The experimental results show that DiPri can improve the performance of the host fuzzers in covering code compared to their default seed prioritization. Future works include refining the designs and implementations of DiPri and conducting more in-depth evaluation.

ACKNOWLEDGEMENT

The authors would like to thank the anonymous reviewers for their insightful comments. This work is supported partially by the National Natural Science Foundation of China (62141215, 62272220), and Science, Technology and Innovation Commission of Shenzhen Municipality (CJGJZD20200617103001003).

REFERENCES

[1] AFL. Accessed, 2023-04-30. American Fuzzy Lop Github Repository. https://github.com/google/AFL.

[2] AFL++Team. Accessed, 2023-04-30. American Fuzzy Lop Plus Plus (afl++). https://github.com/AFLplusplus/AFLplusplus.

[3] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *NDSS*, Vol. 19. 1–15. https://doi.org/10.14722/ndss.2019.23371

[4] Davide Balzarotti. 2021. The Use of Likely Invariants as Feedback for Fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*. https://www.usenix.org/conference/usenixsecurity21/presentation/fioraldi

[5] Marcel Böhme, Valentin JM Manès, and Sang Kil Cha. 2020. Boosting Fuzzer Efficiency: An Information Theoretic Perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE'20)*. 678–689. https://doi.org/10.1145/3368089.3409748

[6] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*. 2329–2344. https://doi.org/10.1145/3133956.3134020

[7] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2017. Coverage-based Greybox Fuzzing As Markov Chain. *IEEE Transactions on Software Engineering (TSE'17)* 45, 5 (2017), 489–506. https://doi.org/10.1145/2976749.2978428

[8] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the Reliability of Coverage-Based Fuzzer Benchmarking. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE'22)*. 1621–1633. https://doi.org/10.1145/3510003.3510230

[9] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. 2020. MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs. In *29th USENIX Security Symposium (Security'20)*. 2325–2342. https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hongxu

[10] Jinfu Chen, Yiming Wu, Chengying Mao, Tsong Yueh Chen, and Haibo Chen. 2021. MMFC-ART: a Fixed-size-Candidate-set Adaptive Random Testing approach based on the modified Metric-Memory tree. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 248–259. https://doi.org/10.1109/qrs54544.2021.00036

[11] Tsong Yueh Chen, Fei-Ching Kuo, Robert G Merkel, and TH Tse. 2010. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software* 83, 1 (2010), 60–66. https://doi.org/10.1016/j.jss.2009.02.022

[12] Tsong Yueh Chen, TH Tse, and Yuen-Tak Yu. 2001. Proportional Sampling Strategy: A Compendium and Some Insights. *Journal of Systems and Software (JSS'01)* 58, 1 (2001), 65–81. https://doi.org/10.1016/s0164-1212(01)00028-0

[13] Cliff Chow, Tsong Yueh Chen, and TH Tse. 2013. The ART of divide and conquer: An innovative approach to improving the efficiency of adaptive random testing. In *2013 13th International Conference on Quality Software*. IEEE, 268–275. https://doi.org/10.1109/qsic.2013.19

[14] 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)*. https://doi.org/10.1145/3478520

[15] 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. https://doi.org/10.1145/3460319.3464795

[16] Shan-Shan Hou, Chun Zhang, Dan Hao, and Lu Zhang. 2013. PathART: Path-sensitive adaptive random testing. In *Proceedings of the 5th Asia-Pacific Symposium on Internetware*. 1–4. https://doi.org/10.1145/2532443.2532460

[17] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. BEACON: Directed Grey-Box Fuzzing with Provable Path Pruning. In *Proceedings-2022 IEEE Symposium on Security and Privacy (SP'22)*. https://doi.org/10.1109/sp46214.2022.9833751

[18] Rubing Huang, Weifeng Sun, Yinyin Xu, Haibo Chen, Dave Towey, and Xin Xia. 2019. A Survey on Adaptive Random Testing. *IEEE Transactions on Software Engineering (TSE'19)* 47, 10 (2019), 2052–2083. https://doi.org/10.1109/TSE.2019.2942921

[19] Sven Kosub. 2019. A note on the triangle inequality for the Jaccard distance. *Pattern Recognition Letters* 120 (2019), 36–38. https://doi.org/10.1016/j.patrec.2018.12.007

[20] James Kukucka, Luís Pina, Paul Ammann, and Jonathan Bell. 2022. CONFETTI: Amplifying Concolic Guidance for Fuzzers. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE'22)*. 438–450. https://doi.org/10.1145/3510003.3510628

[21] Fei-Ching Kuo, Tsong Yueh Chen, Huai Liu, and Wing Kwong Chan. 2008. Enhancing adaptive random testing for programs with high dimensional input domains or failure-unrelated parameters. *Software Quality Journal* 16 (2008), 303–327. https://doi.org/10.1007/s11219-008-9047-6

[22] 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 (ASE'18)*. 475–485. https://doi.org/10.1145/3238147.3238176

[23] Shaohua Li and Zhendong Su. 2023. Accelerating Fuzzing through Prefix-Guided Execution. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 1–27. https://doi.org/10.1145/3586027

[24] 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 (Security'19)*. 1949–1966. https://www.usenix.org/conference/usenixsecurity19/presentation/lyu

[25] Valentin JM 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 (TSE'19)* 47, 11 (2019), 2312–2331. https://doi.org/10.1109/tse.2019.2946563

[26] Chengying Mao, Xuzheng Zhan, TH Tse, and Tsong Yueh Chen. 2019. KDFC-ART: a KD-tree approach to enhancing Fixed-size-Candidate-set Adaptive Random Testing. *IEEE Transactions on Reliability* 68, 4 (2019), 1444–1469. https://doi.org/10.1109/tr.2019.2892230

[27] Barton P Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of Unix Utilities. *Communications of the ACM (CACM'90)* 33, 12 (1990), 32–44. https://doi.org/10.1145/96267.96279

[28] Stefan Nagy. 2023. Research for Practice: The Fun in Fuzzing. *Commun. ACM* 66, 5 (2023), 48–50. https://doi.org/10.1145/3588045

[29] Hoang Lam Nguyen and Lars Grunske. 2022. BeDivFuzz: Integrating Behavioral Diversity into Generator-based Fuzzing. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE'22)*. 249–261. https://doi.org/10.1145/3510003.3510182

[30] Mohammad Norouzi, David J Fleet, and Russ R Salakhutdinov. 2012. Hamming Distance Metric Learning. In *Advances in Neural Information Processing Systems*, Vol. 25. Curran Associates, Inc.

[31] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: coverage-guided property-based testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 398–401. https://doi.org/10.1145/3293882.3339002

[32] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'19)*. 329–340. https://doi.org/10.1145/3339069

[33] Ruixiang Qian, Yuan Zhao, Duo Men, Yang Feng, Qingkai Shi, Yong Huang, and Zhenyu Chen. 2020. Test recommendation system based on slicing coverage filtering. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 573–576. https://doi.org/10.1145/3395363.3404370

[34] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *23rd USENIX Security Symposium (Security'14)*. 861–875. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/rebert

[35] Christoph Schneckenburger and Johannes Mayer. 2007. Towards the determination of typical failure patterns. In *Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting*. 90–93. https://doi.org/10.1145/1295074.1295091

[36] Dongdong She, Abhishek Shah, and Suman Jana. 2022. Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis. In *Proceedings of 2022 IEEE Symposium on Security and Privacy (SP'22)*. https://doi.org/10.1109/sp46214.2022.9833761

[37] Prashast Srivastava and Mathias Payer. 2021. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th acm sigsoft international symposium on software testing and analysis*. 244–256. https://doi.org/10.1145/3460319.3464814

[38] Harmen-Hinrich Sthamer. 1995. *The automatic generation of software test data using genetic algorithms*. University of South Wales (United Kingdom).

[39] Erik Trickel, Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, and Adam Doupé. 2022. Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 116–133. https://doi.org/10.1109/SP46215.2023.00007

[40] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. In *The Network and Distributed*

System Security Symposium (NDSS'20). 1–17. https://doi.org/10.14722/ndss.2020.24422

[41] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. 2022. One Fuzzing Strategy to Rule Them All. In *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*. 1634–1645. https://doi.org/10.1145/3510003.3510174

[42] Wei-Cheng Wu, Bernard Nongpoh, Marwan Nour, Michaël Marcozzi, Sébastien Bardin, and Christophe Hauser. 2023. Fine-Grained Coverage-Based Fuzzing. *ACM Transactions on Software Engineering and Methodology* (2023). https://doi.org/10.1145/3587158

[43] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. 2019. Slf: Fuzzing without Valid Seed Inputs. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE'19)*. 712–723. https://doi.org/10.1109/icse.2019.00080

[44] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In *29th USENIX Security Symposium (Security'20)*. 2307–2324. https://www.usenix.org/conference/usenixsecurity20/presentation/yue

[45] Xiaogang Zhu and Marcel Böhme. 2021. Regression Greybox Fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS'21)*. 2169–2182. https://doi.org/10.1145/3460120.3484596

[46] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *ACM Computing Surveys (CSUR'22)* (2022). https://doi.org/10.1145/3512345

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