Cybersecurity

RESEARCH Open Access

PosFuzz: augmenting greybox fuzzing with effective position distribution

Yanyan Zou[1,2,4,5], Wei Zou[1,2,4,5], JiaCheng Zhao[3,6], Nanyu Zhong[1,2,4,5], Yu Zhang[1,2,4,5], Ji Shi[1,2,4,5] and Wei Huo[1,2,4,5*]

Abstract
Mutation-based greybox fuzzing has been one of the most prevalent techniques for security vulnerability discovery and a great deal of research work has been proposed to improve both its efficiency and effectiveness. Mutation-based greybox fuzzing generates input cases by mutating the input seed, i.e., applying a sequence of mutation operators to randomly selected mutation positions of the seed. However, existing fruitful research work focuses on scheduling mutation operators, leaving the schedule of mutation positions as an overlooked aspect of fuzzing efficiency. This paper proposes a novel greybox fuzzing method, PosFuzz, that statistically schedules mutation positions based on their historical performance. PosFuzz makes use of a concept of effective position distribution to represent the semantics of the input and to guide the mutations. PosFuzz first utilizes Good-Turing frequency estimation to calculate an effective position distribution for each mutation operator. It then leverages two sampling methods in different mutating stages to select the positions from the distribution. We have implemented PosFuzz on top of AFL, AFLFast and MOPT, called Pos-AFL, -AFLFast and -MOPT respectively, and evaluated them on the UNIFUZZ benchmark (20 widely used open source programs) and LAVA-M dataset. The result shows that, under the same testing time budget, the Pos-AFL, -AFLFast and -MOPT outperform their counterparts in code coverage and vulnerability discovery ability. Compared with AFL, AFLFast, and MOPT, PosFuzz gets 21% more edge coverage and finds 133% more paths on average. It also triggers 275% more unique bugs on average.

Keywords Greybox fuzzing, Mutation position, Mutation operator, Code coverage, Vulnerability discovery

*Correspondence:
Wei Huo
huowei@iie.ac.cn
[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[3] State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[4] Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences, Beijing, China
[5] Beijing Key Laboratory of Network Security and Protection Technology, Beijing, China
[6] Zhongguancun Laboratory, Beijing, China

Introduction

Mutation-based greybox fuzzing (https://lcamtuf.coredump.cx/afl/; https://llvm.org/docs/LibFuzzer.html; https://honggfuzz.dev/; Li et al. 2018; Liang et al. 2018; Manès et al. 2019; Serebryany 2017), is one of the most prevalent methods for discovering vulnerabilities in modern software. In particular, mutation-based greybox fuzzing organizes its fuzzing loop into three pipeline stages. *Seed selection* selects a seed from a seed test cases pool, which is initialized by user-provided input cases and updated accordingly in the fuzzing process. The selected seed is assigned an `energy` factor based on its length, improved coverage and execution time. *Input seed mutation* iteratively mutates the selected seed by applying one or more randomly picked mutation operators on several random mutation positions using a `mutation scheduler`.

It generates input test cases according to `energy` of the seed. *Testing* feeds the generated inputs to the target program, detects its abnormal behavior, and records proper runtime information, i.e., code coverage, to provide hints for seed pool update and input mutation.

Related work

Numerous optimization methods spanning seed selection and input seed mutation have been proposed to improve the performance of fuzzing, i.e., the ability to discover vulnerabilities or obtain a higher code coverage of the target program.

Seed selection Fruitful research has focused on the seed selection phase, either adopting better selection strategies (Böhme et al. 2016; Gan et al. 2018; Rawat et al. 2017; Böhme et al. 2017; Lemieux and Sen 2018; Petsios et al. 2017; Chen et al. 2018; She et al. 2022; Herrera et al. 2021), or leveraging different runtime information (Xu et al. 2017; Liang et al. 2018; Nagy and Hicks 2019; Chen et al. 2019; Zong et al. 2020; Yun et al. 2018). AFL (https://lcamtuf.coredump.cx/afl/) and Libfuzzer (https://llvm.org/docs/LibFuzzer.html) leverage a priority queue to schedule input seeds during fuzzing. They both rely on edge coverage as a metric to determine the priority of the input seed and update the seed pool. AFLFast (Böhme et al. 2016) optimizes the scheduling of the input seed by using a Markov chain model to determine the `energy` score of each seed. AFLGo (Böhme et al. 2017) allocates more energy to seeds closer to the target locations and vice versa, which is called a power scheduler. CollAFL (Gan et al. 2018) introduces a novel path-sensitive seed selection policy, which is guided by precise coverage statistics feedback and can efficiently mitigate path collisions. Vuzzer (Rawat et al. 2017) prioritizes input seeds that are unlikely to trigger error-handling basic blocks while Angora (Chen and Chen 2018) prefers ones that are likely to trigger conditional statements with unexplored branches.

Input seed mutation The generation of highly effective input cases is vital for fuzzing performance. This line of research (Rawat et al. 2017; Lemieux and Sen 2018; Chen and Chen 2018; Li et al. 2017; Aschermann et al. 2019; You et al. 2019; Lyu et al. 2019; Wang et al. 2017; Rajpal et al. 2017) assigns mutation operators with different weights by mining the semantics of input bytes. Both Vuzzer (Rawat et al. 2017) and Angora (Chen and Chen 2018) adopt taint analysis to find "magic" bytes in inputs, i.e., bytes that are likely to be referenced in a branch statement. Angora (Chen and Chen 2018) further proposes to solve the path constraints problem using a gradient descent-based search. REDQUEEN (Aschermann et al. 2019) identifies a relationship called "input-to-state correspondence", which makes it possible for some input bytes to be directly mapped to program memory. It leverages this mapping relationship to implement a close but lightweight approximation to taint tracking. GREYONE (Gan et al. 2020) follows the traditional taint analysis to guide the fuzzing process but proposes a novel data flow analysis to tune the fuzzing direction further. PATA (Liang et al. 2022) also conducts path-aware taint analysis to identify input bytes that will be propagated to variables used in path constraints and mutates them accordingly. In summary, taint analysis is vital for identifying critical bytes of inputs. However, it suffers from path explosion and overhead issues, making it hard to scale to large programs or inputs.

There also exist other advancements (You et al. 2019; Lyu et al. 2019; Rajpal et al. 2017) requiring no heavy static/dynamic analysis. AFL contains a simple position-sensitive policy: it skips the bytes that trigger no execution path in the *flip*8 operator for following operators like *flip*16, *flip*32, and arithmetic. However, it is limited to the deterministic stage of AFL, making it not applicable for operators in the havoc stage. ProFuzzer (You et al. 2019) predefines six classes of data fields, adopts a lightweight probing approach to classify input bytes, and performs mutation according to probed data type semantics. MOPT (Lyu et al. 2019) focuses on the scheduling of mutation operators by prioritizing operators that lead to new code coverage, paying no attention to the mutation position schedule. Rajpal et al. (2017) also explores the inequality of input seed bytes regarding effectiveness in triggering interesting cases. It trains an offline LSTM (Sundermeyer et al. 2012) model to capture patterns in past fuzzing and guide future mutations accordingly.

To conclude, existing research still results in generating lots of redundant input cases. On the one hand, if certain mutation operators are selected yet applied to all positions like (Lyu et al. 2019), mutations are performed unnecessarily in the input positions representing raw data, e.g., pixel data of *JPEG*. On the other hand, if the mutation operators are applied selectively to the positions according to the inferred type of data stored in the positions or the predefined weights of mutation operators, e.g., (You et al. 2019; Rajpal et al. 2017), they are not effective on the data with an unknown type.

Testing Orthogonal to the aforementioned two aspects, program instrumenting (Schumilo et al. 2017; Xu et al. 2017; Nagy and Hicks 2019; Andronidis and Cadar 2022) and test scheduling (Zong et al. 2020; Chen et al. 2019) are also proposed to accelerate the testing process. kAFL (Schumilo et al. 2017) extends hardware-assisted feedback fuzzing to OS kernels by designing novel operating system-level primitives. Xu et al. (2017) proposes to solve the file system contention and the scalability of fork() system call under multi-core scenarios. UnTracer (Nagy and Hicks 2019) traces only coverage-increasing test cases to reduce fuzzing overhead. FuzzGuard (Zong et al. 2020) filters out unreachable inputs before executing the target program to boost the performance of fuzzing. EnFuzz (Chen et al. 2019) adopts ensemble learning to fuzzing through cooperatively combining multiple base fuzzers.

Our approach

In this paper, we propose PosFuzz, an augmented greybox fuzzer with a mutation strategy sensitive to the mutation position. PosFuzz requires no prior about the data types of mutation positions. PosFuzz constructs a statistical model which records the historical performance of mutation positions. Furthermore, PosFuzz applies mutation operators on different mutation positions according to the statistical model, reducing redundant input cases. To be specific, PosFuzz computes an effective position distribution for each mutation operator on the fly. Given a mutation operator, an effective position associated with it (abbreviated as an effective position) is a position in the input seed on which the mutation could lead to a new program coverage. PosFuzz schedules mutation positions for a mutation operator using weights denoted by their probabilities in the distribution. Note that the effective position distribution for each mutation operator is calculated regardless of the input format, which makes PosFuzz input format agnostic and more general.

In the input seed mutation phase, PosFuzz first records profiles for each mutation operator, which are pairs of mutation operators and mutation positions if a mutation generates an interesting input case, i.e., an input leading to a new program coverage. Then, it uses online Good-Turing frequency estimation to calculate the position distribution for each mutation operator based on the profiles. At last, it utilizes acceptance-rejection sampling and alias sampling, both of which are guided by the distribution to schedule the positions in two different mutation stages.

We have implemented PosFuzz on top of three state-of-the-art fuzzers, i.e., AFL, AFLFast, and MOPT, called Pos-AFL, -AFLFast, and -MOPT, respectively. Evaluation using 20 real-world programs (UNIFUZZ Li et al. 2021) and LAVA-M dataset shows that, under the 24 h testing time budget, PosFuzz outperforms AFL, AFLFast, and MOPT in code coverage and vulnerability discovery. Compared with its counterpart, Pos-AFL, -AFLFast and -MOPT increases coverage by 22%, 23%, and 17% on average, respectively, finds 131%, 165%, 102% more paths, and triggers 243%, 311%, 271% more unique bugs.

In summary, this paper makes the following contributions:

- We introduce a concept of effective position distribution to represent the semantics of the input, instead of the data type of the positions in the input. The distribution can be used to select the mutation positions with different weights, which can increase the ability to generate interesting input cases.
- We propose PosFuzz, which leverages online Good-Turing frequency estimation to calculate the position distribution for each mutation operator, and schedules the positions based on the distribution for the later mutation.
- We implement PosFuzz on top of AFL, AFLFast, and MOPT. Evaluations using 20 widely used open-source programs and the LAVA-M dataset show that PosFuzz outperforms these three methods in both code coverage and the ability of bug finding.

Background and motivation

We briefly introduce mutation-based greybox fuzzing and its de facto standard, American Fuzzy Lop (AFL) (https://lcamtuf.coredump.cx/afl/). We leverage a widely-used open-source program *exiv*2 to demonstrate the limitation of existing mutation-based greybox fuzzing and the idea of PosFuzz. Specifically, we select an image with type *JPEG* as the input of *exiv*2 to illustrate the problem in this section.

Mutation-based greybox fuzzing

Mutation-based greybox fuzzing, like AFL and its descendants, utilizes edge coverage or bug-finding feedback from the prior execution to guide the fuzzing process and generate more effective inputs. Algorithm 1 presents the overall workflow of AFL. Mutation-based greybox fuzzing commonly requires a user-provided input corpus serving as an initial set of input seeds (Line 1). Usually, the input seeds are organized into a priority queue. The core fuzzing loop (Lines 2–21) repeatedly selects an input seed from the seed queue, mutates it to one or more input cases, and feeds the generated input cases to the target program. The interesting input cases, i.e., those trigger a new code coverage and are added to the priority queue at the suitable position. Thus, the main workflow of mutation-based greybox fuzzing can be divided into three phases:

Phase 1: Seed Selection. An input seed is picked from the priority queue. And several input cases are generated from the input seed based on its energy score. Thus, the key challenge is designing proper priority schemes and scoring methods to rank/score input seeds. A great deal of previous work has explored different strategies to improve the effectiveness of this process. For example, AFLFast leverages the Markov chain to calculate the energy score of each input seed.

Phase 2: Input Seed Mutation. The given input seed is mutated to generate several input cases by performing a series of predefined *mutation operators* on the selected *mutation positions*. The mutation operators define how to manipulate the input seed while the mutation positions denote the places to mutate. AFL partitions this phase into three stages:

- *Deterministic Stage.* A mutation operator with the deterministic feature is selected one by one from the set of predefined mutation operators and applied to every byte of the input seed sequentially (Lines 4–9). For example, *bitflip 1/1 operator* is used first to flip each bit of the input seed, and *bitflip 8/8 operator* is then applied to flip each byte.
- *Havoc Stage.* A mutation operator suitably used in this stage is randomly selected from predefined mutation operators and applied to randomly chosen positions of the input seed (Lines 10–20).
- *Splicing Stage.* The stage is triggered when the previous two stages find no new crashes or paths for all seeds, which rarely happens. In this stage, two input seeds are spliced into a new one, and the havoc stage is re-executed. Algorithm 1 doesn't cover this stage as it performs no mutation.

Algorithm 1: AFL Workflow

```
1  $queue = initialSeed()$;
2  while $TRUE$ do
       /* Phase 1: Seed Selection                                  */
3      $input = selectFromQueue(queue)$;
       /* Phase 2: Input Seed Mutation                             */
       /* Stage 1: Deterimistic Stage                              */
4      foreach $op \in deterministicMutations$ do
5          for $pos \leftarrow 0$ to $len(input)$ do
6              $newInput = mutate(input, op, pos)$;
               /* Phase 3: Testing                                 */
7              $runTest(newInput)$;
8          end
9      end
       /* Stage 2: Havoc Stage                                     */
10     $score = performanceScore(input)$;
11     for $i \leftarrow 0$ to $score$ do
12         $numMutations = havocStacking()$;
13         $newInput = input$;
14         for $j \leftarrow 0$ to $numMutations$ do
15             $op = randomMutation(havocMutations)$;
16             $pos = randomSelect(len(newInput))$;
17             $newInput = mutate(newInput, op, pos)$;
18         end
           /* Phase 3: Testing                                     */
19         $runTest(newInput)$;
20     end
21 end
22 Function RunTest($input$):
23     $runResults = run(input)$;
24     if $isInteresting(runResults)$ then
25         queue.add($input$);
26     end
27 return
```

Note that AFL adopts a uniform mutation strategy. Different mutation positions are selected with equal probability, i.e., a byte definitely selected in the deterministic stage or randomly selected with the same probability in the havoc stage. In addition, the mutation position is manipulated by a mutation operator selected from the predefined operator set with the same probability too. Such a strategy is generally inefficient, leading to generating redundant input cases, which will be explained specifically in the next subsection.

Phase 3: Testing. This phase feeds the input cases to the instrumented target program. If any abnormal behavior or a new code coverage appears, the corresponding input case is marked as interesting and added to the priority queue.

Insights in input seed mutation

We use the *exiv*2 program from UNIFUZZ (Li et al. 2021) benchmark as a motivating example. The *exiv*2 is a cross-platform library for manipulating the metadata of images. Thus, we test *exiv2* for 24 h using *JPEG* images and one state-of-the-art fuzzer AFLFast (Böhme et al. 2016), which shares the same fuzzing process shown in Algorithm 1. We have analyzed the relation between mutation positions/operators and generated interesting cases in detail. We finally obtain two key observations that guide the design of PosFuzz.

Observation 1 Mutation positions are not equal with respect to their efficacy in generating interesting input cases.

Lines 5 and 16 of Algorithm 1 indicate that all positions are assumed with the same importance. Thus, the positions are selected either one by one in the deterministic stage or with equal probability in the havoc stage. However, the input to a target program is usually structured and semantic-rich, making the assumption held by AFL and its descendants doesn't hold for most cases.

Figure 1 illustrates how a *JPEG* image is organized. In summary, the input bytes can be categorized into four kinds:

- Magic Number (blue boxes), marking the image type ([$0x00$, $0x01$]) or header type ([$0x02$, $0x03$] and[$0x14$, $0x15$]).
- Size (green boxes), determining the length of each header ([$0x16$, $0x17$]).
- Enumeration (grey boxes), containing pre-defined discrete values like Identifier ([$0x06$, $0x0A$]), which is a string value "JFIF\0" for *JPEG* format.
- Raw image data (white boxes), representing raw pixel data of the image ([$0x18$, $0x1F$]).

Clearly, the former three types play a more critical role for image metadata manipulator like *exiv*2 than the last raw image data types. For example, a value of $0xFFD8$ in position [$0x00$, $0x01$] will lead *exiv*2 to call the handler corresponding to *JPEG* format while another value like $0x4D4D$ will redirect the target program to *TIFF* handler, which will definitely increase the edge coverage of the target program.

The effectiveness of all positions in a sample *JPEG* file is summarized in Fig. 2. Specifically, we collect the interesting cases associated with the mutation positions, according to Lines 24–26 of Algorithm 1. The horizontal axis of Fig. 2 represents the first 32 bytes of mutation positions (1 byte per slot), while the orange bars against the vertical axis stands for the ratio between interesting input cases and all input cases generated by mutating on this position. We can observe that the interesting input cases are not uniform-distributed over the mutation positions.

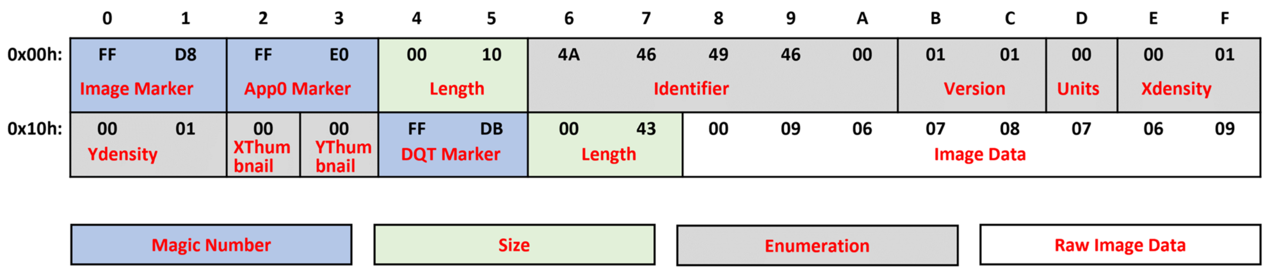

Fig. 1 Sample *JPEG* image

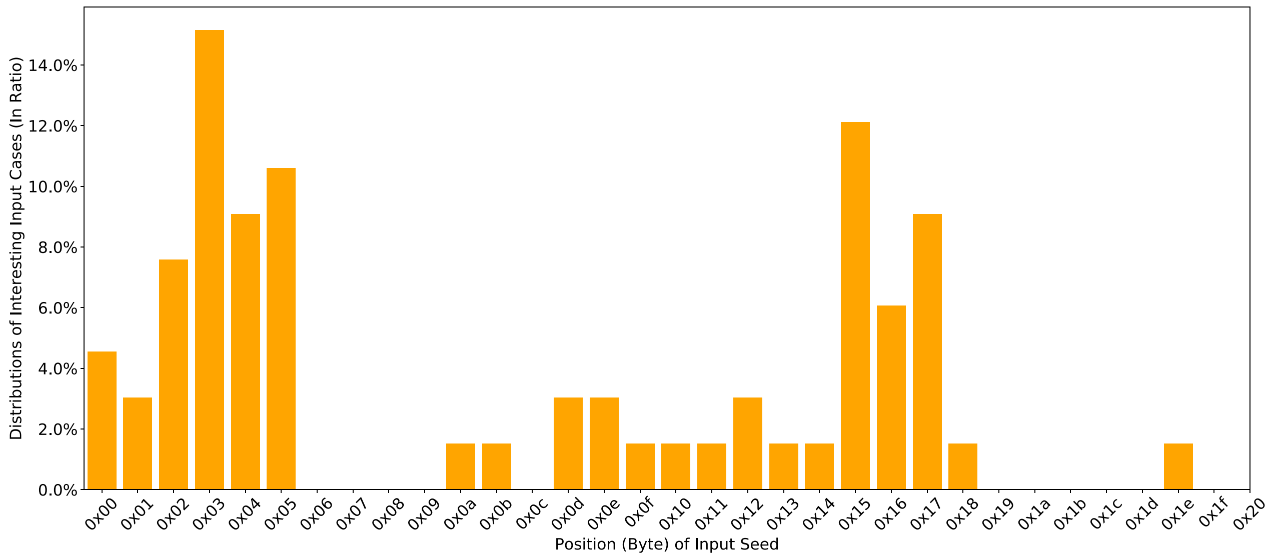

Fig. 2 Effectiveness of the positions for *JPEG* in ratio

Taking position [$0x18$, $0x1F$] for instance, less than 1% of interesting input cases are generated when mutating on these positions as they belong to the Raw image field, which is unlikely to trigger more edges of *exiv*2. However, for position range [$0x02$, $0x03$], an order of magnitude more interesting input cases (about 7% and 15% respectively) are generated with the same amount of mutations, due to the fact that this is a Magic number field, which requires the target *exiv*2 program to perform branching on values in these positions. Thus, our observation indicates that too many mutations are allocated to fewer effective positions (e.g., position range [$0x18$, $0x1F$]).

Observation 2 The efficacy of mutation positions in generating interesting input cases is sensitive to mutation operators.

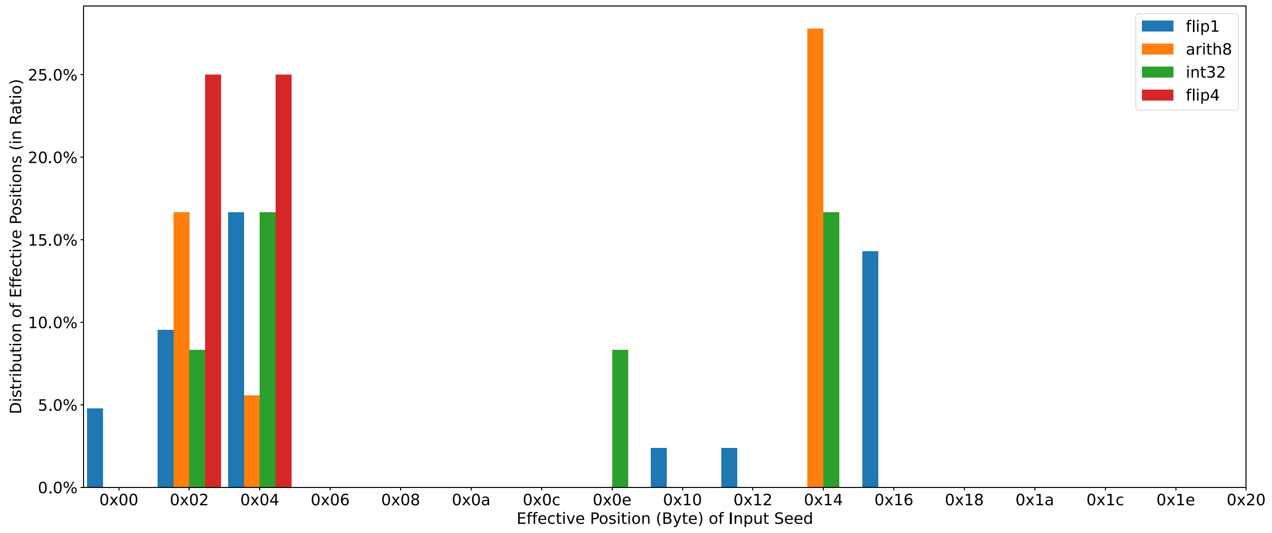

Fig. 3 Sensitivity of mutation positions w.r.t mutation operators in ratio

Line 4 of Algorithm 1 indicates that all the mutation operators in *deterministicMutations* set are assumed with the same importance, as they are applied one by one. Meanwhile, Line 15 of Algorithm 1 indicates that all the mutation operators in *havocMutations* set also are assumed to have the same importance, as they are selected with the same probability. However, we have further discovered that different mutation positions are often interpreted as different semantics, which means the efficacy of the mutation positions is sensitive to the mutation operators.

Figure 3 depicts the sensitivity of mutation positions (2 bytes per slot) with respect to the mutation operators using a histogram denoting the ratio of interesting input cases. The horizontal axis is the positions of input seeds ranging in [$0x00$, $0x1E$] and the vertical axis is the distribution of interesting input cases generated by mutation operators in ratio. The bars with different colors represent four representative mutation operators: *flip1, arith8, int32, flip4*.

Figure 3 shows that the efficacy of mutation positions differs from one mutation operator to another. When applying mutation operator *flip*1, more than 17% interesting input cases are generated by applying the mutation operator to the position [$0x04$, $0x05$], while only 5% interesting input cases are generated by applying mutation operator *arith*8 to these positions. For mutation operator *arith*8, more than 28% interesting input cases are generated by applying to the position [$0x14$, $0x15$], it generates the most interesting input cases compared with other positions.

Key idea of PosFuzz

Based on the observations above, we need to address the problem of selecting suitable positions for certain mutation operators. Instead of inferring the data type of the positions to apply predefined mutation operators for the data type, which is explored by ProFuzzer (You et al. 2019), we treat the selection problem as an online lightweight scheduling problem.

Given a set of mutation operators, we schedule the positions to maximize the probability of generating interesting input cases. The scheduling decision is made based on the effective position distribution of that operator, which records the relative probabilities of generating interesting input cases for all positions. PosFuzz records the historical performance of mutation positions in a per-program and per-operator manner, and calculates effective position distribution on the fly according to the recorded historical performance.

We design PosFuzz, a position-sensitive mutation scheduling method, to augment the current fuzzing techniques without prior knowledge of the input data types. PosFuzz utilizes a statistical model to calculate

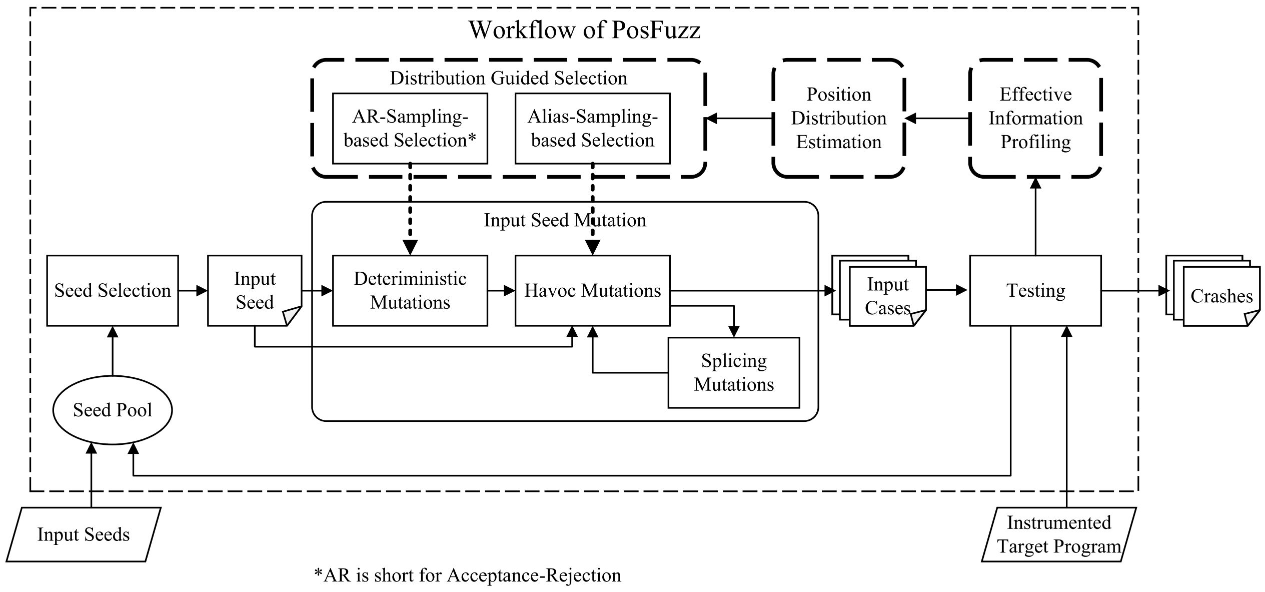
Workflow of PosFuzz
Distribution Guided Selection
AR-Sampling-based Selection*
Alias-Sampling-based Selection
Position Distribution Estimation
Effective Information Profiling
Input Seed Mutation
Seed Selection
Input Seed
Deteriministic Mutations
Havoc Mutations
Splicing Mutations
Input Cases
Testing
Crashes
Seed Pool
Input Seeds
Instrumented Target Program
*AR is short for Acceptance-Rejection

Fig. 4 PosFuzz overview

the effective position distribution of an operator based on the runtime information collected online, making PosFuzz lightweight and more general. The statistical model indicates which positions should be mutated and how much energy should be spent on them for a given mutation operator. An enhanced Good-Turing frequency estimation learns the statistical model based on a given operator's historical performance of mutation positions.

The constructed statistical model further guides position selection. For a given mutation operator, PosFuzz utilizes two sampling methods, i.e., acceptance-rejection sampling for the deterministic stage and alias sampling for the havoc stage, to aid the selection of proper positions for later mutating. The two sampling methods are adopted as they are efficient, i.e., fast in selecting a position, and precise, i.e., the sampled positions follow the calculated distribution.

The statistical model is updated periodically as the fuzzing process mines more semantics of the input types. Thus, PosFuzz divides the whole fuzzing loop into a sequence of *epochs*, each of which is the time unit for the mutation scheduler. During an epoch, PosFuzz learns a statistical model, selects the mutation positions guided by the model, and then gathers the runtime information for updating the model for the next epoch. In such a way, PosFuzz can apply the mutation operators to as many appropriate mutation positions as possible, improving the fuzzing effect.

Approach overview

PosFuzz leverages a position-sensitive scheduler to augment mutation-based greybox fuzzers. The scheduler consists of three main phases, i.e., effective information profiling, position distribution estimation, and distribution-guided selection, on top of the typical greybox fuzzing process. The overall approach is depicted in Fig. 4 with three phases highlighted in bold dotted squares.

PosFuzz divides the mutation process into multiple sequential *epochs*, each *epoch* lasts for a constant time whose value is configurable before fuzzing. PosFuzz records effective information in a per-epoch manner. Information from all previous *epochs* is accumulated to estimate the effective position distribution.

Effective information profiling It records the historical performance of conducted mutations if they trigger interesting input case generation. The effective information is recorded using a pair of a mutation operator and its mutation position, denoted as (op, pos) when an interesting input case is generated during testing. At the end of each *epoch*, the effective information of the current epoch is accumulated to the historical performance data.

For instance, if an interesting input 'id : 001451, src : 0 00931, op : $flip4$, pos : 60' is generated from the source seed #000931 by mutating the position 60 with the mutation operator *flip*4, we record the effective information as ('flip4', 60), and update the history performance accordingly.

Position distribution estimation At the beginning of an *epoch*, the effective position distribution for each mutation operator is re-calculated according to the historical performance. The position distribution of an operator represents the probability of each position being selected for mutation, and it is constructed from effective information like ('flip4', 60), i.e., more input cases generated means a higher selection probability in the future. The critical challenge is to predict the effectiveness of unseen positions precisely, as we are facing a code start problem. We can not assign a zero probability to positions generating no interesting input cases since this position can either be ineffective or not be selected. PosFuzz leverages Good-Turing frequency estimation to tackle this problem. Good-Turing frequency estimation is a statistical method that can provide a simple estimate of the total probability of the objects seen and unseen (Gale and Sampson 1995).

Taking fuzzing *convert* with *TIFF* format input as an example. At the beginning of 25-th epoch, the effective position distribution of operator *flip1* has been re-computed by Good-Turing frequency estimation according to collected data from the previous 24 epochs. The probability of selecting position 54 for mutation is 20.83%, and that probability for position 72 is 10.01%.

Distribution guided selection The distribution is used to guide the input seed mutation phase. Due to the different selecting actions in deterministic mutations and havoc mutations, two sampling methods are employed. For mutation operators in the deterministic stage, the input positions are dealt with one by one, and each position is selected using acceptance-rejection sampling with a probability computed according to the position distribution of the operator. For mutation operators in the havoc stage, a position is chosen from all the input potions using alias sampling with the probability recorded in the distribution for the selected mutation operator.

Given the operator *flip1* and its effective position distribution, PosFuzz can select the mutation positions at a specific probability. For deterministic stage mutations, position 54 is selected at a probability of 100% as it has the most possibility to generate interesting cases among all the positions, while position 72 is selected at a probability of 48.1%. For havoc stage mutations, position 54 and position 72 are chosen at probabilities of 20.83% and 10.01%, respectively, according to the distribution.

Design and implementation

In this section, we formalize the effective position distribution and explain how this distribution helps to select proper positions for a given mutation operator. Note that PosFuzz divides the whole mutation process into multiple sequential *epochs*. Thus, we discuss the key phases in an *epoch*. We also present the algorithm of PosFuzz on top of a typical greybox fuzzing process.

Effective information profiling

Firstly, for each input case, $\mathcal{C}$, PosFuzz records its *linkage*, i.e., how $\mathcal{C}$ is mutated directly from its parent input, as $\mathcal{L}(\mathcal{C})$. The *linkage* of a case is a set of pairs (op, pos). Each pair (op, pos) stands for applying a mutation operator op on a mutation position pos. Note that the order of applying mutation operators is not recorded, as it does not affect the distribution calculation in this work.

Second, for the i-th *epoch*, denoting as $\mathcal{E}_i$, all its interesting cases are put into an input case set, denoting as $IntC(\mathcal{E}_i)$. An input case $\mathcal{C}$ is interesting as long as it can cover a new execution path of the target program.

Position distribution estimation

At the very beginning of each *epoch*, PosFuzz calculates the effective position distribution using profiling information accumulated from all previous *epochs* so far. This section formalizes the definition of effective position distribution for a mutation operator and introduces Good-Turing frequency estimation to compute it.

Effective position distribution definition

The effective position distribution is defined as:

Definition 1 Given a mutation operator op, an effective position distribution $f_{op}(pos)$ for op is a discrete probability function that maps positions to probabilities, and the probability of a position is determined by the number of interesting input cases, the linkage of which contains this position.

For the i-th effective position, its probability means the possibility of being selected for mutation. The probability is denoted as $P_{op}(pos = pos_i)$, and we have

$$\sum_i P_{op}(pos = pos_i) = 1$$

Intuitively, AFL and its variants use a uniform effective position distribution, which indicates that every position is assumed to be equally important regarding generating interesting cases. In this work, for each mutation operator op, $f_{op}(pos)$ is calculated based on the effective historical information profiled from all previous *epochs* and guides the mutation position selection in current *epoch* accordingly.

Good-turing frequency estimation

One straightforward way to estimate the effective position distribution is to directly compute it using effective historical information, i.e., all positions leading to interesting cases are assigned with non-negative probability, while all other positions are assigned zero probability. However, this is not a good way due to the random nature of fuzzing. A position triggering no interesting cases can be either ineffective or given no chance to be selected. It poses a unique challenge for us to distinguish these two types of positions. To tackle this problem, we introduce Good-Turing frequency estimation 0 to smooth the distribution among the unseen positions.

Step 1: Preparing effective information Firstly, all the interesting input cases generated so far are denoted as:

$$U = \text{IntC}(\mathcal{E}_0) \cup \ldots \cup \text{IntC}(\mathcal{E}_{cur})$$

where $\mathcal{E}_{cur}$ represents the current *epoch*.

Then, for a given operator op, all the input cases $\mathcal{C}$ whose linkage contains op denoted as:

$$C(op) = \{\mathcal{C} \mid (op, *) \in \mathcal{L}(\mathcal{C}) \text{ and } \mathcal{C} \in U\}$$

where $(op, *) \in \mathcal{L}(\mathcal{C})$ means that op is at least in one element of $\mathcal{L}(\mathcal{C})$.

Step 2: Calculating $f_{op}(pos)$. We need to estimate the contribution of each position to its corresponding input cases, i.e., the weight of each position. In general, an input case is generated either in *deterministic* stage or in *havoc* stage, and we consider that all input cases are of the same importance. For *deterministic* stage, the input case is generated by applying the mutation operator op to one specific position. For *havoc* stage, the input case is generated by applying random mutation operators to multiple positions. Thus, we need to distribute the weight of the input case to positions in *havoc* stage. Given an interesting case $\mathcal{C}_k$ and an effective pair $(op, pos_i) \in \mathcal{L}(\mathcal{C}_k)$, the weight of the effective position pos_i for the case is defined as:

$$weight_k(pos_i) = \frac{REPEAT_{max}}{|(\mathcal{L}(\mathcal{C}_k))|}$$

where

$$REPEAT_{max} = \max(|\mathcal{L}(\mathcal{C})|) \quad where \quad \forall\, \mathcal{C} \in C(op)$$

Without loss of generality, we assume $REPEAT_{max}$ is divisible by $|(\mathcal{L}(\mathcal{C}))|$ here, as they are both some powers of two in practice.

Given an operator op, the overall weight of an effective position pos_i is defined as:

$$R_i = \sum_k weight_k(pos_i)$$

The overall scaled weight of an effective position pos_i is called the frequency of pos_i.

Given a frequency r, the number of positions with this frequency is denoted as:

$$N_r = |\{pos \mid pos = pos_i \text{ and } R_i = r\}|$$

Then the total number of samples is defined as:

$$N = \sum_{r=0}^{\infty} rN_r$$

According to Good-Turing frequency estimation, a smoothed frequency of r is estimated as:

$$r^* = (r+1)\frac{N_{r+1}}{N_r}$$

Therefore, $f_{op}(pos)$ is calculated as follows. For an observed position pos_i, its corresponding probability is computed as:

$$P_{op}(pos = pos_i) = \frac{r^*}{N} = \frac{(R_i+1)}{N}\frac{N_{R_i+1}}{N_{R_i}}$$

For an unseen position pos_i, its corresponding probability is computed as:

$$P_{op}(pos = pos_i) = \frac{N_1}{N}$$

Distribution guided selection

PosFuzz selects mutation positions after an operator op is selected with the help of a distribution $f_{op}(pos)$. In particular, the position selection is orthogonal to operator selection strategies proposed by other works. As illustrated in "Section Background and motivation", two distinct stages, i.e., *deterministic* and *havoc* stages exist in de-facto mutation-based greybox fuzzing frameworks. Thus, two different selection strategies are designed accordingly.

Augmenting deterministic stage

In *deterministic* stage, traditional mutation-based greybox fuzzing applies all possible operators one by one, and each operator is performed on all possible positions, leading to an equal selecting probability (the probability is one) for each position. Instead, given an operator, PosFuzz selects positions with different probabilities. For a position pos_i, PosFuzz leverages *Acceptance-Rejection Sampling* (Flury 1990) method to decide whether to mutate on it with an acceptance probability. The probability is defined as:

$$Pacc_{op}(pos = pos_i) = \frac{P_{op}(pos = pos_i)}{\max(P_{op}(pos = pos_j))} \tag{1}$$

Augmenting Havoc stage

In *havoc* stage, a traditional fuzzer like AFL selects mutation operators randomly and chooses mutation positions for each operator using a uniform distribution, i.e., selecting positions with the same probability. Different from that, PosFuzz incorporates the *Alias Sampling* (Shapiro and Silverman 1960) method, an efficient sampling method, to choose some mutation position in the input case with the probability recorded in $f_{op}(pos)$. *Alias Sampling* is a general method to sample from a discrete probability distribution. It requires $O(nlogn)$ or $O(n)$ preprocessing time and $O(1)$ time to select a position based on $f_{op}(pos)$. Thus, in this stage, a position is selected with a probability defined as:

$$Psel_{op}(pos = pos_i) = P_{op}(pos = pos_i) \tag{2}$$

PosFuzz implementation

Algorithm 2 illustrates a typical implementation of PosFuzz, which is on top of AFL.

The fuzzing process consists of loops of *epochs*, each of which can last for a user-configured time interval, e.g., 1 h. During an *epoch*, appropriate mutation positions are selected based on the effective position distribution computed from the effective historical information.

Algorithm 2: PosFuzz Workflow

```
queue = initialSeed();
newEpoch = TRUE, history = [], δ = [];
while TRUE do
    input = selectFromQueue(queue);
    if newEpoch then
        /* Update effective information and Estimation Position
           Distribution                                                  */
        history = history + δ;
        dist = goodTuringEstimation(history);
        newEpoch = FALSE;
    end
    foreach op ∈ deterministicMutations do
        for pos ← 0 to len(input) do
            /* Accept the pos for op via Acceptance-Rejection Sampling   */
            accepted = ifAcceptPos(dist, op, pos);
            if accepted then
                newInput = mutate(input, op, pos);
                runTest(newInput, (op, pos)));
            end
        end
    end
    score = performanceScore(input);
    for i ← 0 to score do
        numMutations = havocStacking();
        newInput = Input, linkage = [];
        for j ← 0 to numMutations do
            op = randMutation(havocMutations);
            /* Select pos for op via Alias Sampling                      */
            pos = selectPosForOp(dist, op, len(newInput));
            newInput = mutate(newInput, op, pos);
            linkage.add(op, pos)
        end
        runTest(newInput, linkage);
    end
    /* Update runtime of current epoch                                   */
    if epoch_runtime > epoch_maxtime then
        newEpoch = TRUE;
    end
end
Function runTest(input, linkage):
    runResults = run(input);
    if isInteresting(runResults) then
        queue.add(input);
        δ += linkage;
    end
return
```

At the beginning of an *epoch*, PosFuzz accumulates the effective information of the current epoch and updates it to the history information *history* (Line 6), which will be used to calculate the effective distribution. PosFuzz utilizes Good-Turing frequency estimation to compute the effective distribution *dist* based on the effective historical information *history* profiled so far (Line 7). It then selects positions with the help of the distribution. In deterministic stage, it leverages *Acceptance-Rejection Sampling* method, represented as *ifAcceptPos*, to determine whether the current position should be selected at a probability of $Pacc_{op}(pos = pos_i)$ (Line 12). In havoc stage, it adopts *Alias Sampling* method, represented as *selectPosForOp*, to get a random position with a probability of $Psel_{op}(pos = pos_i)$ (Line 25).

Posfuzz records the *linkage* of an operator in the format of (*op*, *pos*) (Lines 15 and 27) when mutating. If an interesting case is generated, the *linkage* is added to the interesting pair δ of the current epoch (Line 39).

Evaluation

To demonstrate the generality and effectiveness of PosFuzz, we implement three variants of PosFuzz, Pos-AFL, Pos-AFLFast, and Pos-MOPT, atop of three state-of-the-art greybox fuzzers, i.e., AFL (https://lcamtuf.coredump.cx/afl/), AFLFast (Böhme et al. 2016) and MOPT (Lyu et al. 2019), respectively. We choose AFL due to its popularity in production. AFLFast and MOPT are selected due to they are orthogonal to PosFuzz. AFLFast proposes a novel seed selection policy while MOPT augments AFL with efficient scheduling of mutation operators.

Evaluation dataset We evaluate PosFuzz using UNIFUZZ benchmark (Li et al. 2021) and a standard dataset LAVA-M. UNIFUZZ contains 20 real-world programs categorized into six functionality types, i.e., *image, audio, video, text, binary, and network*. The setup of UNIFUZZ benchmark is in line with (Li et al. 2021), and the second column of Table 1 lists all the 20 programs. We report edge coverage and unique executing paths for UNIFUZZ benchmark. Moreover, both UNIFUZZ and LAVA-M are utilized to evaluate PosFuzz regarding bug discovery ability.

Experiment settings We run six fuzzing tools, i.e., AFL/Pos-AFL, AFLFast/Pos-AFLFast, and MOPT/Pos-MOPT, for each program in the dataset. For each program with each fuzzing tool alternative, we run the experiment five times with the same seeds and each run lasts 24 h. We report average results across the five runs. Notice the test time for Pos-AFL, -AFLFast, and -MOPT (Pos-* for short) contains effective distribution calculation and position sampling, i.e., all the runtime overhead is included. A 1-h epoch is picked across all experiments for Pos-*. All experiments are conducted

Table 1 The average edge coverage for PosFuzz on AFL, AFLFast and MOPT

No.	Program	AFL	Pos-AFL	INC	AFLFast	Pos-AFLFast	INC	MOPT	Pos-MOPT	INC
1	exiv2	5.53	14.09	155%	8.02	14.17	77%	8.09	13.35	65%
2	tiffsplit	4.57	5.49	20%	4.65	5.63	21%	4.72	5.25	11%
3	mp3gain	4.22	4.60	9%	3.97	4.61	16%	4.14	4.60	11%
4	wav2swf	0.62	0.62	1%	0.62	0.65	6%	0.63	0.65	3%
5	pdftotext	15.34	18.01	17%	14.97	18.05	21%	14.86	17.87	20%
6	infotocap	2.54	3.38	33%	2.41	3.61	50%	2.95	3.36	14%
7	mp42aac	3.57	3.98	11%	3.51	4.04	15%	4.11	4.35	6%
8	flvmeta	0.78	0.80	2%	0.75	0.80	7%	0.79	0.80	1%
9	objdump	6.65	8.60	29%	6.72	8.75	30%	6.44	8.67	35%
10	tcpdump	9.85	16.27	65%	9.72	15.82	63%	9.89	16.48	67%
11	ffmpeg	27.85	28.74	3%	28.32	29.85	-1%	28.13	28.58	2%
12	gdk-pixbuf-pixdata	2.95	3.81	29%	2.92	4.23	45%	2.95	3.96	34%
13	cflow	2.84	2.98	5%	2.82	2.99	6%	2.84	2.99	5%
14	nm-new	6.50	7.60	17%	6.15	7.76	26%	6.42	7.89	23%
15	sqlite3	33.43	37.15	11%	32.43	37.24	15%	32.44	36.37	12%
16	lame3.99.5	9.50	9.64	1%	9.18	9.63	5%	9.40	9.64	3%
17	jhead	0.52	0.52	0%	0.52	0.52	0%	0.52	0.52	0%
18	imginfo	3.67	4.84	32%	3.43	4.90	43%	3.97	4.71	19%
19	jq	5.97	6.24	5%	5.90	6.23	6%	6.15	6.26	2%
20	mujs	9.00	9.20	2%	8.21	9.24	13%	8.48	9.12	8%
	Average			22%			23%			17%

on a two-socket, 80 cores machine with Intel Xeon Gold 6248v4@2.50GHz processor.

Effectiveness of augmentation for fuzzing real-world programs

We first report results on UNIFUZZ benchmark, which contains 20 real-world programs.

The overall results

Coverage and paths Tables 1 and 2 depict the average edge coverage and paths for all six fuzzers. In particular, column *Program* denotes the target programs and column *INC* illustrates the improvement of Pos-* over its counterpart, i.e., Pos-AFL over AFL.

Pos-* outperforms its counterpart in edge coverage for 58 out of 60 cases. On average, Pos-* improves edge coverage by 22% for AFL, 23% for AFLFast, and 17% for MOPT. Moreover, Pos-* explores 131%, 165%, and 102% more paths than AFL, AFLFast, and MOPT on average, respectively. Note that the results of Pos-MOPT show that the fine-grained mutation position selection on top of the coarse-grained mutation operator selection (what exactly MOPT does) can still eliminate the redundant input cases, thus exploring more paths and getting higher coverage.

Unique bugs We leverage the number of unique bugs found as a metric of bug discovery ability for fuzzing tools. We extract the top three functions in the stack trace from the output of ASan (Serebryany et al. 2012) to de-duplicate bugs, and divide the bugs that have different stack trace and vulnerability types as unique bugs. Table 3 presents the total number of unique bugs triggered by each fuzzer in five repetitions on the 20 real-world programs. Pos-* triggers more unique bugs for all of the programs. On average, Pos-AFL, Pos-AFFast, and Pos-MOPT detect 243%, 311%, and 271% more unique bugs than their counterparts. In particular, PosFuzz identifies 10x more unique bugs for the *tcpdump* program due to the significant improvement in edge coverage and paths.

Detail comparison

Figures 5 and 6 depict the detailed experiment results of fuzzing efficiency for the first fuzzing execution, in terms of edge coverage and execution paths. Each small figure is associated with a target program, in which the green dotted line, green line, red dotted line, red line, blue dotted line, and blue line represent AFL, Pos-AFL, AFLFast, Pos-AFLFast, MOPT, and Pos-MOPT, respectively.

Edge coverage For each small figure in Fig. 5, the horizontal axis is the edge coverage in ratio, and the vertical axis is the hours lasting for fuzzing. As shown in Fig. 5, Pos-* outperforms its counterpart for all input formats and target programs, showing good generality.

Table 2 The average paths for PosFuzz on AFL, AFLFast and MOPT

No.	Program	AFL	Pos-AFL	INC	AFLFast	Pos-AFLFast	INC	MOPT	Pos-MOPT	INC
1	exiv2	246.6	1741.4	606%	523.2	1982.4	279%	476.2	1458.8	206%
2	tiffsplit	989.8	1746.2	76%	1151.6	1948.8	69%	1315.2	1655.2	26%
3	mp3gain	1265.8	1945	54%	1040.4	1945.4	87%	1239.6	1854	50%
4	wav2swf	224.2	228.2	2%	230.2	293.2	27%	256.6	293.6	14%
5	pdftotext	4237.4	8212.8	94%	3726	7918.8	113%	3610.8	8163.4	126%
6	infotocap	1218.2	3119.4	156%	1177.6	3525.4	199%	2644.8	2948.6	11%
7	mp42aac	502.8	1186.6	136%	482.4	1174.4	143%	1715	1967	15%
8	flvmeta	525.6	678.6	29%	478.6	691.8	45%	579.8	648.6	12%
9	objdump	866.4	2470.4	185%	964	2604.2	170%	774.8	2463.4	218%
10	tcpdump	1450.8	5332.4	268%	1364.6	5062.6	271%	1522	5419.6	256%
11	ffmpeg	600.2	1066.4	78%	352	1176.4	234%	369.2	872.6	136%
12	gdk-pixbuf-pixdata	188.2	699.2	272%	172.6	972.2	463%	181.6	833	359%
13	cflow	500	892.4	78%	453.8	912.8	101%	514.6	948.4	84%
14	nm-new	1571.6	2548.6	62%	1167.8	2905.2	149%	1531.8	2809.4	83%
15	sqlite3	3187.2	8751.8	175%	2161.8	8891.2	311%	2282.6	7434.8	226%
16	lame3.99.5	1493.8	2335	56%	1274.4	2330.2	83%	1443.2	2300.8	59%
17	jhead	293.6	362.8	24%	258.4	374	45%	350.8	364.8	4%
18	imginfo	453.4	1154	155%	276.8	1280.8	363%	585.2	1093.2	87%
19	jq	906.6	1543.4	70%	938.6	1593.6	70%	1522.4	1550.6	2%
20	mujs	4380.2	6324	44%	3698	6214	68%	3738.4	5800.4	55%
	Average			131%			165%			102%

Table 3 The total number of unique bugs in five repetitions for PosFuzz on AFL, AFLFast and MOPT

No.	Program	AFL	Pos-AFL	AFLFast	Pos-AFLFast	MOPT	Pos-MOPT
1	exiv2	5	11	3	7	5	6
2	tiffsplit	10	14	11	15	11	16
3	mp3gain	5	7	4	7	4	7
4	wav2swf	2	3	3	3	3	3
5	pdftotext	1	2	0	2	0	3
6	infotocap	7	10	5	15	10	10
7	mp42aac	0	0	0	0	0	0
8	flvmeta	2	2	2	2	2	2
9	objdump	8	14	4	14	3	14
10	tcpdump	12	134	11	134	12	132
11	ffmpeg	0	0	0	0	0	0
12	gdk-pixbuf-pixdata	0	7	0	16	0	16
13	cflow	0	2	0	3	0	3
14	nm-new	0	0	0	0	0	0
15	sqlite3	0	1	1	1	0	3
16	lame3.99.5	3	5	3	4	3	4
17	jhead	5	5	5	5	5	5
18	imginfo	0	0	0	0	0	0
19	jq	2	2	2	2	2	2
20	mujs	3	4	3	4	2	4
	Total	65	223	57	234	62	230

Execution path For each small figure in Fig. 6, the horizontal axis is the number of execution paths, and the vertical axis is the hours lasting for fuzzing. As shown in Fig. 5, Pos-* outperforms their counterparts dramatically for most of the 20 target programs.

Specifically, Pos-* significantly improve the path discovery for some image formats, e.g., *imginfo* and *jpg*. The effectiveness of positions varies widely across image formats due to their difference in semantic specifications and structured data layout. Thus, PosFuzz achieves better performance when Pos-* acquires more accurate and valuable effective position distribution information from runtime profiling, e.g., *imginfo* and *jpg*. However, for some audio and video formats, e.g., *wav* and *mp*4, the effective positions distribute relatively even, so Pos-* do not explore much more paths.

In summary, PosFuzz can automatically use statistical information to guide the selection of mutation positions without perceiving program semantics or input formats, which can improve both edge coverage and explore more execution paths.

Evaluation on LAVA-M

LAVA-M consists of 4 target programs and is widely adopted as a standard benchmark for measuring the vulnerability discovery abilities of fuzzers. The initial seed sets and all other settings are in line with (Rawat et al. 2017), and they are identical across all six fuzzers evaluated. For each program, we run the experiment five times, and each run lasts 24 h.

Table 4 lists accumulated bugs in 5 runs for all fuzzers. Pos-* outperforms its counterpart significantly. Pos-AFL detects 533 more bugs than AFL, Pos-AFLFast detects 469 more bugs than AFLFast, and Pos-MOPT detects 126 more bugs than MOPT. In particular, Pos-* finds all listed bugs in *who*, and 4 unlisted bugs in addition. Moreover, for the most buggy program *who*, Pos-* finds 466, 418, 118 more listed bugs and 42, 32, 8 more unlisted bugs than AFL, AFLFast, and MOPT, respectively. Moreover, only Pos-AFLFast can identify one bug on *md*5*sum* while all other fuzzers fail to find any.

Case study: *exiv2*

In this section, we leverage the program *exiv*2, a cross-platform library for manipulating metadata of images to investigate how PosFuzz outperforms state-of-the-art fuzzing tools like AFLFast (Böhme et al. 2016). We test *exiv*2 for 24 h using both AFLFast and Pos-AFLFast, the setting of execute parameter and initial seeds are in line with UNIFUZZ.

Intuitively, high-level semantics are crucial for improving fuzzing performance. Take image formats for instance, positions containing metadata, e.g., the header positions (Magic number in "Section Insights in input seed mutation")

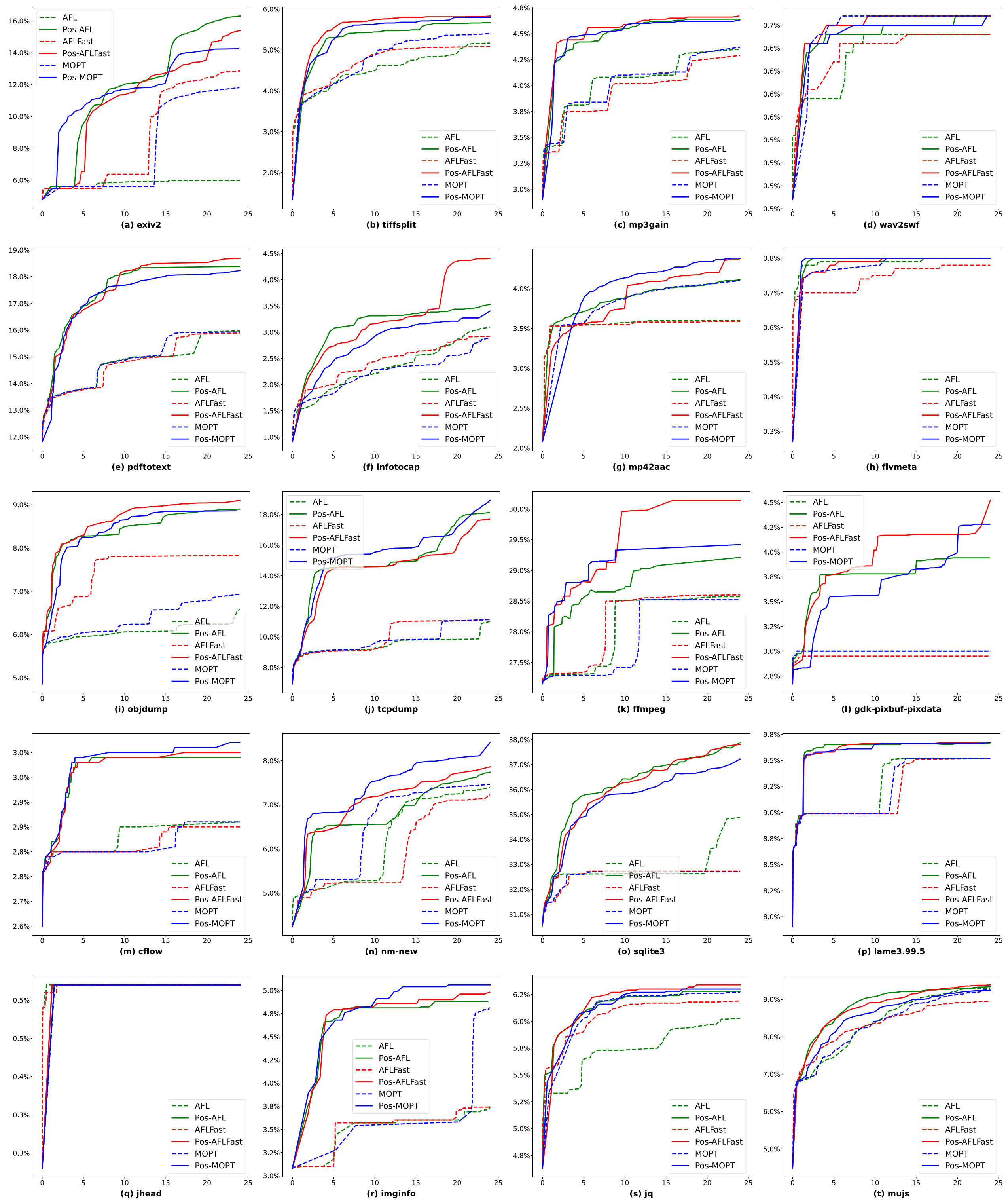

Fig. 5 Comparison of edge coverage

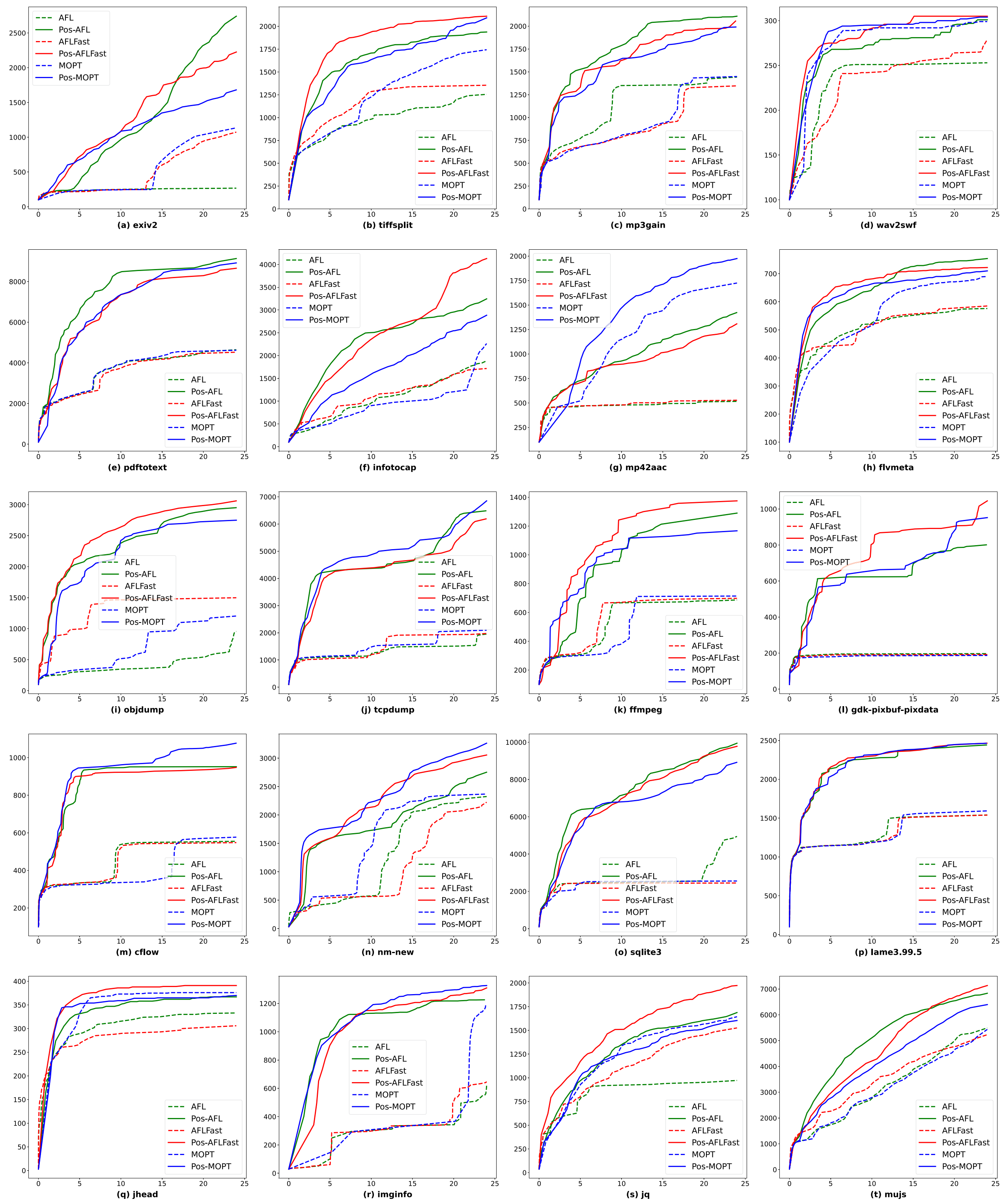

Fig. 6 Comparison of paths

determining whether an image is *JPEG* or *TIFF*, are more likely to increase edge coverage, compared to positions containing only pixel data (Raw image data in "Section Insights in input seed mutation"). However, instead of directly probing the types of input bytes as adopted by ProFuzzer (You et al. 2019), PosFuzz chooses to conduct scheduling in a utility-guided fashion: scheduling positions according to coverage improvement instead of scheduling them using their semantic types. This design makes PosFuzz type-agnostic and requires no pre-defined types.

```
void main(BasicIo::AutoPtr io)
{
    ...
    if (isJpegType(image)){
        readJpegMeta(image);
    }
    if (isTiffType(image)){
        readTiffMeta(image);
    }
    ...
}
bool isJpegType(Image& image){
    image.read(tmpBuf, 2);
    if (0xff != tmpBuf[0]|| JpegImage::soi_ != tmpBuf[1])
        result = false;
    ...
}

bool isTiffType(Image& image){
    image.read(tmpBuf, 8);
    if (buf[0] == 'I' && buf[0] == buf[1])
        byteOrder_ = littleEndian
    else if (buf[0] == 'M' && buf[0] == buf[1])
        byteOrder_ = bigEndian;
    else return false;
    ...
}

void JpegBase::readJpegMeta(Image& image)
{
    int marker = advanceToMarker();
    if (marker < 0) throw Error(15);

    while (marker != sos_ && marker != eoi_ && ....) {
        image.read(tmpBuf, minSize);
        uint16_t size = getUShort(tmpBuffer, bigEndian);
        if ( marker == app1_ && .....) {
            if (size < 8)
                rc = 1; break;
            ...
        }
        else if ( marker == app1_ && ...) {
                if (size < 31)
                    rc = 6; break;
                ...
            }
        else if ( marker == app13_ && ...) {
                if (size < 16)
                    rc = 2; break;
                ...
            }
        else if (inRange2(marker,sof0_,sof3_,sof5_,sof15_) && ...){
                if (size < 8)
                    rc = 7; break;
                ...
                image->seek(size-...);
                ...
            }
        else {
            ...
        }
        marker = advanceToMarker();
    }
}
```

Listing 1: Code snippet for *exiv2*.

Table 4 The accumulated bugs triggered on LAVA-M

Program	Listed bugs	AFL	Pos-AFL	AFLFast	Pos-AFLFast	MOPT	Pos-MOPT	Arguments
base64	44	32	44(+4)	37(+1)	44(+4)	44(+4)	44(+4)	-d @@
md5sum	57	0	0	0	1	0	0	-c @@
uniq	28	2	9	2	10	13	13	@@
who	2136	24(+4)	490(+48)	79(+8)	497(+40)	414(+50)	532(+58)	@@
Total	2265	58(+4)	543(+52)	118(+9)	552(+44)	471(+54)	589(+62)	–

Listing 1 is a code snippet extracted from *exiv*2. *exiv*2 first determines types of input images by querying position 0*x*00 with two-byte length: value 0*xFFDB* for *JPEG* (Line 14) and value 0*x*4*D*4*D* or 0*x*4949 (ASCII code for letter 'II' or 'MM') for *TIFF* (Lines 21 and 22). Manipulating position [0*x*00, 0*x*01] will clearly lead to different paths in *exiv*2 as every image type has its own handler,e.g., *isJpegType* for *JPEG* and *isTiffType* for *TIFF*. Moreover, given that an image is *JPEG* format, the *exiv*2 further examines position 0*x*02 (`marker` in Line 31) with a 2-byte length and position 0*x*04 (`size` in Line 36). The *exiv*2 will perform a switch-case style branch based on `marker` (from Line 37 to Line 57), and the edge coverage will certainly increase if the fuzzer mutates this `marker` position. The `size` field is used to perform sanity checks according to *JPEG* specification (Line 38, 43, 48 and 53) and directly influences how much data will be read by the program, which may incur out-of-bounds access. Thus, mutating on `size` is likely to trigger more interesting cases. The same analysis holds for position 0*x*14 and 0*x*16, which is another `marker` and `size` pair according to "Section Background and motivation. Besides, the raw image data will be ignored by *exiv*2 (Line 56) as *exiv*2 is a metadata manipulating tool and no further action is performed based on the value of pixel data, making it less effective in increasing edge coverage. In summary, mutation on positions like Magic number and Size is more effective than positions like Raw image data. PosFuzz demystifies the efficacy difference between different positions and assigns more weight to more effective positions using online scheduling based on a statistical model.

Figure 7 depicts the distribution of the interesting cases for both Pos-AFLFast and AFLFast with the x-axis being positions (1-byte slot) and the y-axis being the absolute number of interesting cases. For simplicity reasons, we only report interesting cases that are triggered by *flip1* mutation operator for a position range [0*x*00, 0*x*1*F*], and one interesting case is amortized to all positions when it is generated by mutating on multiple positions, e.g. cases in *havoc* stage.

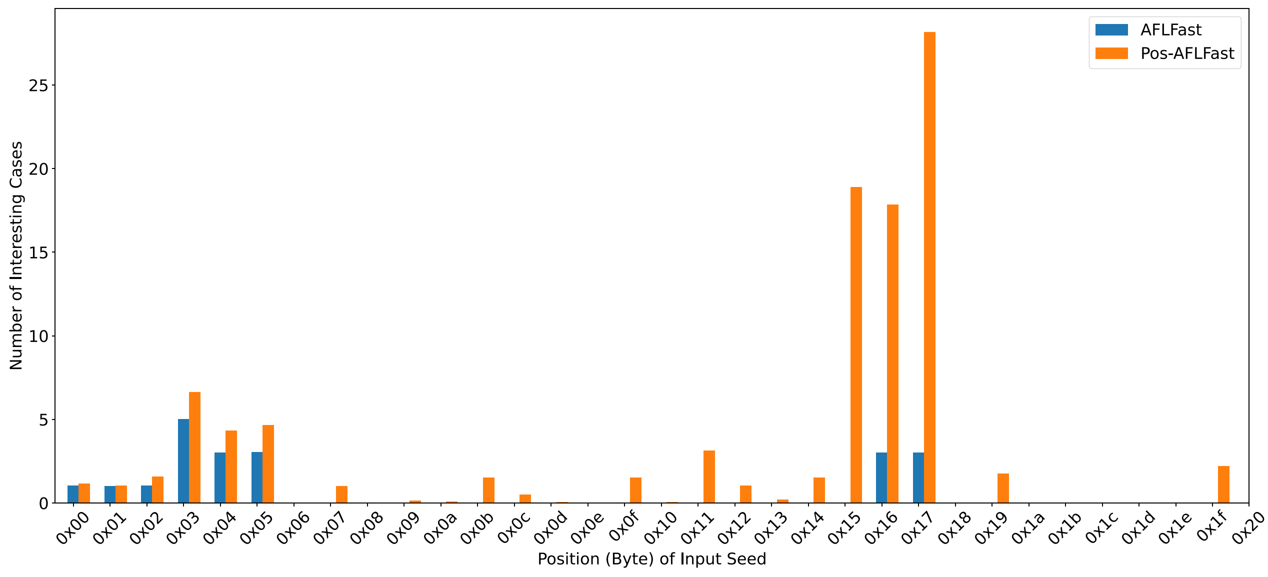

Fig. 7 Histogram of interesting cases of both Pos-AFLFast and AFLFast for *JPEG*

In total, Pos-AFLFast triggers $3.8X$ more interesting cases than AFLFast under the same time budget for *flip1* operator. According to Fig. 7, we can first confirm Observation 1 in "Section Background and motivation holds for *exiv2* program, and only a subset of positions (8 out of 32 for AFLFast and 21 out of 32 for Pos-AFLFast) can trigger interesting cases. Second, compared with AFLFast, we can observe that Pos-AFLFast explores significantly more interesting cases (66 vs 6) in a position range $[0x14, 0x17]$, which is a Marker and Size pair as stated before. AFLFast only finds 6 interesting cases in this position range as it wastes a lot of mutations in less effective positions.

Figure 8 further depicts the distribution of input cases generated for both AFLFast and Pos-AFLFast. The total number of input cases is on par with each other, e.g., 188440 for AFLFast and 183523 for Pos-AFLFast. However, the distribution of inputs across positions differs significantly. AFLFast is unaware of the effectiveness difference between positions, and adopts a uniform distribution, leading to nearly even input cases for each position, as shown by blue bars in Fig. 8. On the contrary, Pos-AFLFast schedules positions according to their historical performance in triggering interesting cases and assigns more mutations to effective positions. For example, 28441 input cases are generated for position $0x17$, while only 635 cases are for position $0x1E$. In summary, PosFuzz schedules mutation positions using a statistical model and assigns more weight to positions like Magic numbers and Size, thus avoiding redundant mutation on positions like Raw image data.

Discussions

In this section, we briefly discuss the relationship between PosFuzz and other fruitful research work in the literature.

First, PosFuzz is orthogonal to methods focusing on seed selection, e.g. AFL (https://lcamtuf.coredump.cx/afl/) and AFLFast (Böhme et al. 2016). PosFuzz can be combined with them transparently, and we have conducted augmentation for AFL and AFLFast, respectively. Second, PosFuzz aims to generate highly effective input cases, which has been explored by numerous research work (Rawat et al. 2017; Li et al. 2017; Chen and Chen 2018; Aschermann et al. 2019; Lyu et al. 2019; You et al. 2019; Wang et al. 2017; Lemieux and Sen 2018; Rajpal et al. 2017). PosFuzz shares the same principle among other research work: leverage the semantics of input bytes to generate more effective input cases (Rawat et al. 2017; Li et al. 2017; Chen and Chen 2018; Aschermann et al. 2019; Lyu et al. 2019; You et al. 2019; Wang et al. 2017; Lemieux and Sen 2018; Rajpal et al. 2017). However, PosFuzz is built upon runtime profiling of interesting test cases with negligible overhead while existing work requires static/dynamic program analysis (Rawat et al. 2017; Chen and Chen 2018; Aschermann et al. 2019; Gan et al. 2020; Liang et al. 2022) which may suffer from the overhead of offline analysis. ProFuzzer (You et al. 2019) requires no static/dynamic program analysis but needs to predefine six classes for input bytes while PosFuzz is input type-agnostic and doesn't need to define the data types beforehand. MOPT (Lyu et al. 2019) prioritizes operators that lead to new code

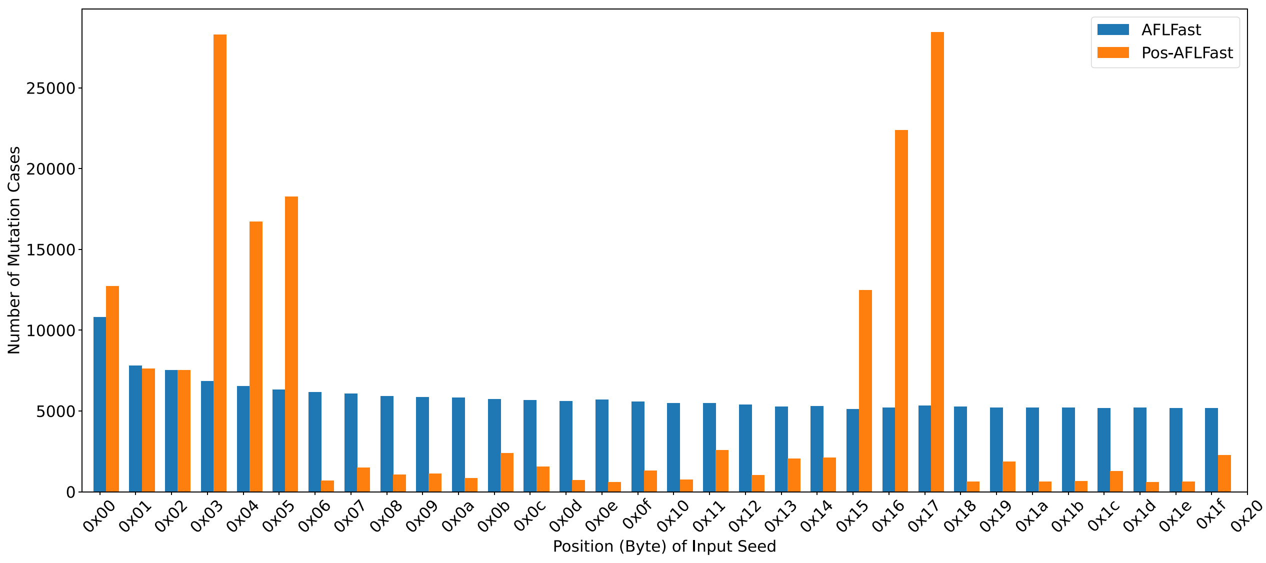

Fig. 8 Histogram of mutation cases of both Pos-AFLFast and AFLFast for *JPEG*

coverage and can be combined with PosFuzz to improve fuzzing performance, as evaluated in "Section Evaluation". Rajpal et al. (2017) schedules mutation positions based on an offline-trained LSTM model while PosFuzz requires no pre-fuzzing training and can adapt to new inputs dynamically, i.e., update the distribution every *epoch*.

Conclusion

We identify the inefficiency of the existing mutation positions scheduler and leverage the effectiveness of different positions for different mutation operators. We propose a mutation position-sensitive scheduler for input mutation, which utilizes a statistical method to direct the selection of mutation positions for a mutation operator. Using the novel scheduler, PosFuzz augments three state-of-the-art fuzzers and implements three prototypes, Pos-AFL, -AFLFast, and -MOpt. We evaluated these fuzzers on the UNIFUZZ benchmark (20 widely used open source programs) and LAVA-M dataset, and the evaluation result shows that Pos-* outperform their counterparts in both edge coverage and vulnerability discovery.

Acknowledgements
Not applicable.

Author contributions
YYZ and WH designed this research. YYZ, JCZ, NYZ built this framework and performed experiments. YYZ and WH wrote this paper. YZ, JS, WZ reviewed and edited the manuscript. All authors read and approved the manuscript.

Funding
This research was supported by National Key R&D Program of China (2022YFB3103900), and National Natural Science Foundation of China (62032010, 62202462), and Strategic Priority Research Program of the CAS (XDC02030200).

Availability of data and materials
All public dataset sources are as described in the paper.

Declarations

Competing interests
The authors declare that they have no competing interests.

Received: 6 August 2022 Accepted: 10 February 2023
Published online: 01 June 2023

References

A Security Oriented, Feedback-driven, Evolutionary, Easy-to-use Fuzzer with Interesting Analysis Options. https://honggfuzz.dev/
American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/
Andronidis A, Cadar C (2022) Snapfuzz: high-throughput fuzzing of network applications
Aschermann C, Schumilo S, Blazytko T, Gawlik R, Holz T (2019) Redqueen: fuzzing with input-to-state correspondence. In: NDSS, vol 19, pp 1–15
Böhme M, Pham V-T, Roychoudhury A (2016) Coverage-based greybox fuzzing as markov chain. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp 1032–1043
Böhme M, Pham V-T, Nguyen M-D, Roychoudhury A (2017) Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, pp 2329–2344
Chen P, Chen H (2018) Angora: Efficient fuzzing by principled search. In: 2018 IEEE symposium on security and privacy (SP). IEEE, pp 711–725
Chen Y, Jiang Y, Ma F, Liang J, Wang M, Zhou C, Jiao X, Su Z (2019) Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In: 28th USENIX Security Symposium (USENIX Security 19), pp 1967–1983
Chen H, Xue Y, Li Y, Chen B, Xie X, Wu X, Liu Y (2018) Hawkeye: towards a desired directed grey-box fuzzer. In: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security, pp 2095–2108
Flury BD (1990) Acceptance-rejection sampling made easy. SIAM Rev 32(3):474–476
Gale WA, Sampson G (1995) Good-turing frequency estimation without tears. J Quant Linguist 2(3):217–237
Gan S, Zhang C, Chen P, Zhao B, Qin X, Wu D, Chen Z (2020) GREYONE: Data flow sensitive fuzzing. In: 29th USENIX security symposium (USENIX Security 20), pp 2577–2594
Gan S, Zhang C, Qin X, Tu X, Li K, Pei Z, Chen Z (2018) Collafl: Path sensitive fuzzing. In: 2018 IEEE symposium on security and privacy (SP). IEEE, pp 679–696
Herrera A, Gunadi H, Magrath S, Norrish M, Payer M, Hosking AL (2021) Seed selection for successful fuzzing. In: Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis, pp 230–243
Lemieux C, Sen K (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, pp 475–485
Li J, Zhao B, Zhang C (2018) Fuzzing: a survey. Cybersecurity 1(1):1–13
Liang H, Pei X, Jia X, Shen W, Zhang J (2018) Fuzzing: state of the art. IEEE Trans Reliab 67(3):1199–1218
Liang J, Jiang Y, Chen Y, Wang M, Zhou C, Sun J (2018) Pafl: extend fuzzing optimizations of single mode to industrial parallel mode. In: Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, pp 809–814
Liang J, Wang M, Zhou C, Wu Z, Jiang Y, Liu J, Liu Z, Sun J (2022) Pata: Fuzzing with path aware taint analysis. In: 2022 2022 IEEE symposium on security and privacy (SP). IEEE Computer Society, Los Alamitos, CA, USA, pp 154–170
LibFuzzer - a Library for Coverage-guided Fuzz Testing. https://llvm.org/docs/LibFuzzer.html
Li Y, Chen B, Chandramohan M, Lin S-W, Liu Y, Tiu A (2017) Steelix: program-state based binary fuzzing. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering, pp 627–637
Li Y, Ji S, Chen Y, Liang S, Lee W-H, Chen Y, Lyu C, Wu C, Beyah R, Cheng P et al. (2021) Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In: 30th USENIX security symposium (USENIX Security 21). USENIX Association
Lyu C, Ji S, Zhang C, Li Y, Lee W-H, Song Y, Beyah R (2019) MOPT: Optimized mutation scheduling for fuzzers. In: 28th USENIX security symposium (USENIX security 19), pp 1949–1966
Manès VJM, Han H, Han C, Cha SK, Egele M, Schwartz EJ, Woo M (2019) The art, science, and engineering of fuzzing: a survey. IEEE Trans Softw Eng
Nagy S, Hicks M (2019) Full-speed fuzzing: reducing fuzzing overhead through coverage-guided tracing. In: 2019 IEEE symposium on security and privacy (SP). IEEE, pp 787–802
Petsios T, Zhao J, Keromytis AD, Jana S (2017) Slowfuzz: automated domain-independent detection of algorithmic complexity vulnerabilities. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, pp 2155–2168
Rajpal M, Blum W, Singh R (2017) Not all bytes are equal: neural byte sieve for fuzzing. arXiv preprint arXiv:1711.04596
Rawat S, Jain V, Kumar A, Cojocar L, Giuffrida C, Bos H (2017) Vuzzer: Application-aware evolutionary fuzzing. In: NDSS, vol 17, pp 1–14

Schumilo S, Aschermann C, Gawlik R, Schinzel S, Holz T (2017) kafl: Hardware-assisted feedback fuzzing for OS kernels. In: 26th USENIX security symposium (USENIX Security 17), pp 167–182
Serebryany K (2017) Oss-fuzz-google's continuous fuzzing service for open source software
Serebryany K, Bruening D, Potapenko A, Vyukov D (2012) AddressSanitizer: a fast address sanity checker. In: 2012 USENIX annual technical conference (USENIX ATC 12), pp 309–318
Shapiro HS, Silverman RA (1960) Alias-free sampling of random noise. J Soc Ind Appl Math 8(2):225–248
She D, Shah A, Jana S (2022) Effective seed scheduling for fuzzing with graph centrality analysis. In: 2022 2022 IEEE symposium on security and privacy (SP) (SP). IEEE Computer Society, Los Alamitos, CA, USA, pp 1558–1558
Sundermeyer M, Schlüter R, Ney H (2012) LSTM neural networks for language modeling. In: Thirteenth annual conference of the international speech communication association
Wang J, Chen B, Wei L, Liu Y (2017) Skyfire: Data-driven seed generation for fuzzing. In: 2017 IEEE symposium on security and privacy (SP). IEEE, pp 579–594
Xu W, Kashyap S, Min C, Kim T (2017) Designing new operating primitives to improve fuzzing performance. In: Proceedings of the 2017 ACM SIGSAC conference on computer and communications security, pp 2313–2328
You W, Wang X, Ma S, Huang J, Zhang X, Wang X, Liang B (2019) Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In: 2019 IEEE symposium on security and privacy (SP). IEEE, pp 769–786
Yun I, Lee S, Xu M, Jang Y, Kim T (2018) Qsym: a practical concolic execution engine tailored for hybrid fuzzing. In: 27th USENIX Security Symposium (USENIX Security 18). USENIX Association, Baltimore, MD, pp 745–761
Zong P, Lv T, Wang D, Deng Z, Liang R, Chen K (2020) Fuzzguard: filtering out unreachable inputs in directed grey-box fuzzing through deep learning

Publisher's Note