Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

AMSFuzz: An adaptive mutation schedule for fuzzing

Xiaoqi Zhao, Haipeng Qu [*], Jianliang Xu, Shuo Li, Gai-Ge Wang

College of Computer Science and Technology, Ocean University of China, Qingdao 266100, China

ARTICLE INFO

Keywords:
Fuzzing
Schedule
Multi-armed bandit problem
Path discovery
Bug detection
Vulnerability

ABSTRACT

Mutation-based fuzzing is one of the most popular software testing techniques. After allocating a specific amount of energy (i.e., the number of testcases generated by the seed) for the seed, it uses existing mutation operators to continuously mutate the seed to generate new testcases and feed them into the target program to discover unexpected behaviors, such as bugs, crashes, and vulnerabilities. However, the random selection of mutation operators and sequential selection of mutation positions in existing fuzzers affect path discovery and bug detection. In this paper, a novel adaptive mutation schedule framework, AMSFuzz is proposed. For the random selection of mutation operators, AMSFuzz has the ability to adaptively adjust the probability distribution of mutation operators to select mutation operators. Aiming at the sequential selection of mutation positions, seeds are dynamically sliced with different sizes during the fuzzing process and giving more seeds the opportunity to preferentially mutate, improving the efficiency of fuzzing. AMSFuzz is implemented and evaluated in 12 real-world programs and LAVA-M dataset. The results show that AMSFuzz substantially outperforms state-of-the-art fuzzers in terms of path discovery and bug detection. Additionally, AMSFuzz has detected 17 previously unknown bugs in several projects, 15 of which were assigned CVE IDs.

1. Introduction

Fuzzing is an automatic software testing technique that is proved to be very effective in catching vulnerabilities and bugs. Since it was first proposed by Miller et al. (1990), fuzzing has been widely used in academia and industry due to its simplicity and ease of use. Many companies such as Google (Chris et al., 2011), Adobe (Adobe, 2009), and Microsoft (GitHub, 2020b) have also developed their own fuzzers and used them to discover many vulnerabilities. Many fuzzers such as CollAFL (Gan et al., 2018), LibFuzzer (Serebryany, 2016), UAFL (Wang et al., 2020b), and VUzzer (Rawat et al., 2017) have been developed and improved by researchers.

Mutation-based fuzzing uses multiple mutation operators to generate new testcases and send them to target programs to discover as many program anomalies as possible. AFL (GitHub, 2019a) is a popular mutation-based grey box fuzzer. Most of current fuzzers such as AFLGO (Böhme et al., 2017), AFLTurbo (Sun et al., 2020), and AFLFast (Böhme et al., 2019) are variants of AFL. AFL provides two mutation stages, the deterministic mutation stage and the non-deterministic mutation stage. The former is a sequential selection of mutation locations and mutation operators according to seed size. The latter is a random selection of mutation operators and mutation locations.

However, the random selection of mutation operators in the non-deterministic stage allows each operator to be selected fairly, without considering the effectiveness of different operators and different programs (Lyu et al., 2019). The sequential selection of mutation positions in the deterministic stage allows the selected seed to be mutated sequentially from head to tail. The sequence of mutations makes the number of seed mutations limited by seed size. If the size of a seed is too large, too much time will be spent on that seed, affecting the mutation of other seeds, especially if there are a large number of waiting seeds in the seed queue.

To address these issues, we propose an adaptive mutation schedule framework, named AMSFuzz, which can adaptively select mutation operators according to the target program in the non-deterministic stage and automatically allocate mutation regions by a seed slicing mechanism in the deterministic stage. In the non-deterministic stage, fuzzing is modeled using a multi-armed bandit model. AMSFuzz records the reward of different mutation operators and calculates the probability distribution of mutation operators. Mutation operators are selected according to the calculated probability distribution. In the deterministic stage, a seed slicing mechanism is proposed. AMSFuzz dynamically allocates mutation regions of seeds. After a seed is selected, AMSFuzz determines the start location of the mutation and slicing size to execute mutation.

* Corresponding author.
E-mail addresses: zhaoxiaoqi@stu.ouc.edu.cn (X. Zhao), quhaipeng@ouc.edu.cn (H. Qu), xjl9898@ouc.edu.cn (J. Xu), li_shuo@stu.ouc.edu.cn (S. Li), wgg@ouc.edu.cn (G.-G. Wang).

https://doi.org/10.1016/j.eswa.2022.118162
Received 24 October 2021; Received in revised form 26 May 2022; Accepted 9 July 2022
Available online 16 July 2022

The framework is designed and implemented and named AMSFuzz by extending AFL. AMSFuzz is compared with the popular fuzzers, including AFL, AFLFast, and FairFuzz in terms of path discovery, coverage, and bug detection, and the experiments show that AMSFuzz is more efficient in path discovery, coverage, and bug detection. The effectiveness of our proposed approaches is also evaluated by comparing our fuzzer with MOPT and AFL. Besides, we further evaluate 5 open-source projects on GitHub and Sourceforge platforms. AMSFuzz find 17 bugs, 15 of which are identified by CVE.

In summary, we have made the following contributions.

(1) An adaptive operator selection is proposed and used in the non-deterministic stage to select mutation operators.

(2) A seed slicing mechanism is proposed and used in the deterministic stage, which allows the seeds to adaptively determine the size of mutation regions according to the fuzzing process to improve the efficiency of fuzzing.

(3) The approaches are integrated into AFL and a fuzzer is implemented and named AMSFuzz.

(4) AMSFuzz is evaluated on 12 real-world programs and the LAVA-M dataset. The experiment demonstrates that AMSFuzz has better performance in path discovery and bug detection compared with other fuzzers.

(5) AMSFuzz is used to fuzz open-source programs and find a total of 17 new unknown bugs, 15 of which are assigned CVE IDs.

The rest of this paper is organized as follows.

Section 2 introduces background and related work. Section 3 elaborates the design of AMSFuzz. The evaluation is presented in Section 4. Discussion is shown in Section 5. Section 6 offers the conclusion and future work.

2. Background and related work

In this section, we introduce backgrounds of mutation-based fuzzing, mutation strategies, and multi-armed bandit problem and discuss related work on fuzzing.

2.1. Mutation-based fuzzing

Mutation-based fuzzing is to generate testcases by mutating existing seeds. AFL and its variants are mutation-based grey box fuzzers, which use evolutionary algorithms to continuously mutate seeds to fuzz target programs.

The basic process of AFL is shown in Algorithm 1. The initial seeds provided by users and the instrumented target program are inputs of the algorithm. The seed inputs are used to initialize a seed queue Q (Line 1), AFL enters an infinite loop (Line 3) and it does not exit unless it is manually terminated. In the infinite loop, a seed is first selected from the seed queue Q (Line 4), and energy is allocated to the selected seed (Line 5). The energy determines the number of testcases generated by the seed in the non-deterministic (Lines 11–13) stage. Then, AFL performs deterministic (Lines 6–8) mutations. It sequentially selects a mutation operator and a mutation position according to the length of selected seeds (Line 7) and executes the program (Line 9). Next, the previously calculated energy (Line 5) is used to randomly select mutation positions and mutation operators to mutate and execute the target program (Lines 11–14). After each execution of the target program, AFL determines whether the generated testcase has triggered a crash or discovered a new path and saves it in the crash set C or the seed queue Q (Lines 17–22), respectively.

Algorithm 1: AFL Algorithm.

```
Input: Seed input: $seeds$, Instrumented target program: $P$
Output: Seed queue: $Q$, Crash set: $C$
1  $Q \leftarrow seeds$
2  $C \leftarrow \emptyset$
3  while TRUE do
4      $s \leftarrow$ ChooseNext($Q$)
5      $e \leftarrow$ AssignEnergy($s$)
6      for $op$ in $deterministicOps$ do
7          for $i$ in Range(0, $s.length$) do
8              $s' \leftarrow$ Mutation($s, op, i$)
9              $status \leftarrow$ Run_Target($P, s'$)
10             Save_Update($status$)
11     for $i$ in Range(0, $e$) do
12         $op \leftarrow$ Random($non-deterministicOps$)
13         $s' \leftarrow$ Mutation($op$)
14         $status \leftarrow$ Run_Target($P, s'$)
15         Save_Update($status$)
16 Procedure Save_Update($status$)
17 if is_Crash($status$) then
18     $C \leftarrow C \cup s'$
19     return
20 if is_NewCoverage($status$) then
21     $Q \leftarrow Q \cup s'$
22     return
```

2.2. Mutation strategies

Mutation strategies determine how the seed is mutated and which part of the seed is selected for mutation. Mutation strategies consist of a series of mutation operators. The mutation of AFL is divided into two stages, deterministic stage and non-deterministic stage.

Deterministic stage. The seed is mutated when it is first selected, and it sequentially selects mutation locations from the head of seeds to the tail of seeds to mutate and execute the target program using mutation operators in the deterministic stage.

Mutation operators in the deterministic stage include bitflip, arithmetic, interest, and dictionary.

- The bitflip uses different flip lengths and step lengths to mutate seed. It includes bitflip 1/1, bitflip 2/1, bitflip 4/1, bitflip 8/8, bitflip 16/8, bitflip 32/8 (i.e., flip 32bits per 8-bit step);
- The arithmetic performs integer addition and subtraction operations. It includes arith 8/8, arith 16/8, and arith 32/8. The big-endian and little-endian are also used in arithmetic;
- The interest performs substitution using pre-define interesting values (eg., 0, 1). It includes interest 8/8, interest 16/8, and interest 32/8;
- The dictionary is to overwrite and insert using user-supplied tokens and overwrite auto tokens. It includes user extras (over), user extras (insert), and auto extras (over). We do not provide tokens in this paper.

Non-deterministic stage. The non-deterministic stage includes havoc stage and splice.

- The havoc stage randomly selects mutation positions and mutation operators in deterministic stage to mutate seed.
- The splice cuts each of the two seed files in two parts, and splices the head and tail.

2.3. Multi-armed bandit problem

The multi-armed bandit (MAB) problem is fundamental problem in reinforcement learning, which is a sequential choice problem by defined a series of actions. Multi-armed bandit formulations model decision problems arising in many technical and scientific disciplines, such as economics (Scott, 2015), sensor networks (Li et al., 2020), communication networks (Avner & Mannor, 2019), and clinical trials (Villar et al., 2015).

The MAB problem (Mahajan & Teneketzis, 2008) consists of a MAB process and a controller. At each time, the controller can select exactly one arm, all other arms remain frozen. Each arm i, $i = 1, 2, \ldots, k$ is described by the state $S_i(t)$, and reward $R_i(S_i(t))$, where t denotes the number of times arm has been selected. $S_i(t)$ denotes the state of arm i after it has been select t times. $R_i(S_i(t))$ denotes the reward of arm i in state $S_i(t)$. The objective of the MAB problem is how to select arms sequentially in finite time to maximize total rewards.

The balance between exploration and exploitation is a common problem in the MAB problem (Cuevas et al., 2014). Exploration means making multiple attempts to gain more information to more accurately select better arms for higher rewards, and exploitation means only selecting arms with higher rewards to achieve higher returns after gaining more information. Therefore, our goal is to achieve a trade-off between exploration and exploitation that is constantly and dynamically adjusted between exploration and exploitation in order to achieve higher rewards.

2.4. Related work

Many fuzzing techniques have been proposed and applied in different applications.

Improving the effectiveness of fuzzing.

(1) Seed selection. It controls how a seed is selected from seed queue. AFLFast (Böhme et al., 2019) prefers seeds that have performed low-frequency paths. AFLGO prefers seeds that have reached target locations, MooFuzz (Zhao et al., 2021) uses many-objective optimization (Wang et al., 2020a, 2022; Wang & Tan, 2019) for seed schedule, and MemLock (Cheng et al., 2020) prefers seeds that cause more memory consumption. Cerebro (Li et al., 2019) uses multi-objective algorithm (Rizk-Allah et al., 2017) to balance various metrics to select seeds that have the potential to trigger vulnerabilities.

(2) Power schedule. It determines the energy. AFLFast gives more energy to seeds that reach low-frequency paths, AFLGO supplies more energy to seeds that are closer to target locations. EcoFuzz (Yue et al., 2020) uses an energy-saving approach that models a multi-armed bandit model to perform the energy allocation in havoc stage.

(3) Mutation & schedule. This determines how to mutate and which mutation operators are selected. FairFuzz (Lemieux & Sen, 2018) identifies rare branches and uses a mutation mask algorithm to improving path coverage. MOPT uses PSO to select mutation operators in havoc stage. OTA (Li et al., 2021) uses PSO (Duan et al., 2012) to optimize mutation time in the deterministic stage.

Grammar-aware fuzzing. For highly structured inputs, such as XML and JavaScript, target programs require testcases to satisfy the input specification. Superion (Wang et al., 2019) utilizes two mutation strategies to generate structured testcases. SLF (You et al., 2019) performs a dependency analysis to generate reasonable inputs. Learn&Fuzz (Godefroid et al., 2017) uses neural networks to learn to generate grammatically correct inputs.

Hybrid fuzzing. Hybrid fuzzing is mainly combined with symbolic execution to aid fuzzing. Driller (Stephens et al., 2016) combines the strength of the fuzzer AFL and the symbolic execution engine Angr (Wang & Shoshitaishvili, 2017) to execution. AFL can fast generate testcases and Angr can solve path constraints. QSYM (Yun et al., 2018) uses a dynamic binary instrumentation framework to improve the performance of hybrid fuzzing.

Fuzzing various applications. Many fuzzing researches appear to address specific issues in different areas, such as constraint solvers (Mansur et al., 2020; Winterer et al., 2020), kernels (GitHub, 2021bb; Kim et al., 2020), virtual machines (Schumilo et al., 2020, 2021), smart contracts (He et al., 2019), firmware (Yu et al., 2019), and machine learning models (Xie et al., 2019).

3. AMSFuzz

In this section, we first show the overview of AMSFuzz and then introduce adaptive operator selection and seed slicing mechanisms, respectively.

3.1. Overview

AMSFuzz is designed to address the limitations on fuzzing caused by random selection of mutation operators in the non-deterministic stage and sequential selection of mutation positions in the deterministic stage, improving the performance of fuzzing. Fig. 1 shows a high level AMSFuzz architecture which mainly consists of seed slicing, slicing mutation, exploration vs exploitation, operator selection, and information update.

Exploration vs exploitation, operator selection, and information update are used to select mutation operators in the non-deterministic stage and the details of their design are shown in Section 3.2. AMSFuzz first models a multi-armed bandit model and determines whether the current stage is exploration or exploitation. Then, AMSFuzz selects a mutation operator depending on the current stage. Finally, AMSFuzz updates related information throughout the fuzzing process and calculates the probability distribution before the next mutation.

Seed slicing and slicing mutation are used in the deterministic stage and the details of design are shown in Section 3.3. Seed slicing is mainly to select a mutation region of seeds in which all mutation operators in deterministic stage are used to execute the target program. AMSFuzz first identifies the start location of seed slicing before each mutation and calculates the size of slicing. AMSFuzz then determines mutation regions of seeds and executes mutation.

3.2. Adaptive operator selection

In this subsection, an adaptive operator selection is introduced. We first describe how to model a multi-armed bandit model. Then, we explore how to balance exploration and exploitation. Next, how to select a mutation operator is introduced. Finally, information update and probability calculations are presented.

3.2.1. Adaptive operator selection model

The havoc stage in the non-deterministic stage is taken to model a MAB model. The objective is to choose a mutation operator that maximizes the number of paths. The model is based on two assumptions: (1) the number of paths and crashes generated by target programs is limited; (2) target programs are stateless that means each execution of target programs only depends on mutation operators.

The attributes of our MAB model that correspond to fuzzing are described as follows,

Arms: AMSFuzz employs an arm per operator, and tries to find an optimal operator in havoc stage. Each operator is denoted as op_i, $i = 1, 2, \ldots, k$. k denotes the number of operators in havoc stage.

Action: An action represents the program is fuzzed (i.e., seed mutation and program execution) at once.

Rewards: Rewards refer to the number of discovered paths after the target program is fuzzed many times and are denoted as R. The reward of the target program being fuzzed once is denoted as r. When the target program is fuzzed for the mth time, if the generated testcase produces a new path, the reward r is 1. If no new path, the reward r is 0.

Exploration vs Exploitation: Exploration aims to select a random operator which can collect more information to guide to obtain higher rewards. Exploitation aims to select operators based on a calculated probability distribution that can explore higher rewards.

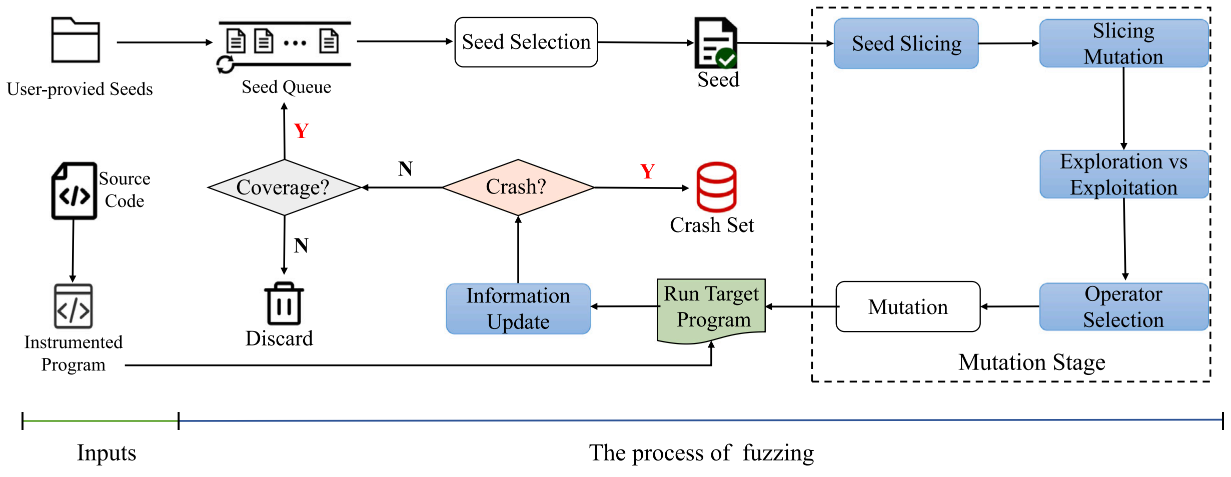

Fig. 1. The overview of AMSFuzz.

3.2.2. Exploration vs exploitation

The balance between exploration and exploitation is a classic problem in the MAB problem. Different from other MAB models, the search space of new path is decreasing with time in the fuzzing process. Fig. 3 illustrates average paths found by different fuzzers in real-world programs within 24 h, which can be demonstrated that the speed of fuzzers discovering new paths is gradually slowing down over time. Because, the paths are finite in the program and the number of new paths increases, the search space of new paths keeps decreasing. AMSFuzz should spend more time performing the exploration stage in this situation. Exploration and exploitation should also be dynamically adjusted as fuzzing process.

Considering that the search space of paths is limited by the fuzzing process, the selection of exploration and exploitation is based on whether new paths are discovered by the previous seed. In our design, the mutation result of the previous seed is used to select exploration and exploitation before the current seed performs the operator selection. If a new path is found after the previous seed performs the mutation, the exploitation stage is selected. Otherwise, AMSFuzz performs the exploration stage.

3.2.3. Operator selection

AMSFuzz performs an operator selection after finishing the selection of exploration and exploitation. It uses a random operator selection in the exploration and a operator selection based on probability distribution in the exploitation.

Operator selection takes into account the fact that fuzzing is an uncertain process. For the program, different mutation operators generate new paths with uncertainty. There is no guarantee that the mutation operators with the most rewards must produce a new path the next time. Therefore, an operator selection based on probability distributions can solve this problem.

The operator selection is shown in Algorithm 2. The number of mutation operators n and the probability of mutation operators $Prob[n]$, mutation operators $ops[n]$, and a stage $stage$ are the inputs, where $Prob[n]$ is calculated by Formula (4), and $stage$ is obtained in Sub Section 3.2.2. In the exploration stage, AMSFuzz generates a random number and uses it to select a mutation operator (Lines 1–3). In the exploitation stage, the probabilities of all mutation operators are first mapped to a continuous space (Lines 5–9), and then a random number is generated to select mutation operators (Lines 10–14).

Algorithm 2: Operator Selection

```
Input: Number of mutation operators: $n$, Probability of
       mutation operators: $Prob[n]$, Mutation operators:
       $ops[n]$, Stage: $stage$
Output: A selected mutation operator $op$
1  if $stage$ is exploration then
2      $i \leftarrow \text{Random}(0, n)$
3      $op \leftarrow ops[i]$
4  else
5      for $i$ in Range$(0, n)$ do
6          if $i \neq 0$ then
7              $Pt[i] \leftarrow Pt[i-1] + Prob[i]$
8          else
9              $Pt[i] \leftarrow Prob[i]$
10     $rand \leftarrow \text{Random}(0, 100)$
11     for $i$ in Range$(0, n)$ do
12         if $rand < Pt[i] \times 100$ then
13             $op \leftarrow ops[i]$
14             break
```

3.2.4. Information update

AMSFuzz generates new testcases to execute the target program by using the selected mutation operator. After each program execution, a set of information is updated to guide the next selection of mutation operators.

Hit count of mutation operators. After the program has been fuzzed $(t+1)$ times, the count of hits $C^{t+1}_{op_i}$ for a mutation op_i is updated as follows,

$$C^{t+1}_{op_i} = \begin{cases} C^{t}_{op_i} + 1 & select_{op_i} = true \\ C^{t}_{op_i} & select_{op_i} = false \end{cases} \tag{1}$$

where $select_{op_i}$ indicates whether the operator op_i is selected.

Rewards of mutation operators. After the program has been fuzzed $(t+1)$ times, the reward $R^{t+1}_{op_i}$ of the selected mutation operator op_i is determined by whether mutation operator op_i produces a new path, which is updated by the following formula,

$$R^{t+1}_{op_i} = \begin{cases} R^{t}_{op_i} + r^{t+1}_{op_i} & found_{op_i} = true \\ R^{t}_{op_i} & found_{op_i} = false \end{cases} \tag{2}$$

where $r_{op_i}^{t+1}$ denotes the reward obtained when the program is fuzzed for the $(t+1)$th time, $found_{op_i}$ indicates whether a new path is generated after selecting the mutation operator op_i.

Effectiveness of mutation operators. After the above information has been updated, the effectiveness calculation of mutation operators is based on the reward of mutation operators $R_{op_i}^{t+1}$ and the hit count of mutation operators $C_{op_i}^{t+1}$. After the program has been fuzzed $(t+1)$ times, the effectiveness $E_{op_i}^{t+1}$ of operator op_i is calculated using the following formula.

$$E_{op_i}^{t+1} = \begin{cases} \frac{R_{op_i}^{t+1}}{C_{op_i}^{t+1}} & C_{op_i}^{t+1} \neq 0 \\ 0 & others \end{cases} \tag{3}$$

Probability of mutation operators. The effectiveness information of all operators is used to calculate the probability $P_{R_{op_i}}^{t+1}$ of an operator op_i being selected. After the program has been fuzzed $(t+1)$ times, the calculation formula is as follows,

$$P_{R_{op_i}}^{t+1} = \begin{cases} \frac{E_{op_i}^{t+1}}{\sum_{i=1}^{K} E_{op_i}^{t+1}} & \sum_{i=1}^{K} E_{op_i}^{t+1} \neq 0 \\ 0 & others \end{cases} \tag{4}$$

where $E_{op_i}^{t+1}$ denotes the effectiveness of operator op_i after the program has been fuzzed $(t+1)$ times, K denotes the number of mutation operators, the value is 16.

3.3. Seed slicing

In this subsection, we present seed slicing to solve the sequential selection of mutation positions in the deterministic stage. We first introduce why seeds are sliced in fuzzing. Then, we describe how to determine the size of slicing and how to mutate the slicing region.

3.3.1. The idea of slicing

The idea of slicing is proposed based on the following considerations. (1) AFL uses sequential mutations in the determination stage. The sequential mutation makes the number of testcases generated by the seed affected by seed length; The longer the seed, the number of testcases generated by the seed will increase. If the capacity of the new path generated by the selected seed is insufficient, it will impede the mutations of other seeds, thereby affecting the whole fuzzing process. (2) Considering the whole fuzzing process, the path search is dynamically changing. As time changes, the search space for paths is gradually reduced, which means that the fuzzer's ability to explore new paths will gradually decrease. However, the sequential mutation uses a fixed mutation length and does not take into account the effect of the search space of paths. Based on the above problems, we propose the idea of slicing. Considering the change of the search space of paths, we hope that the mutation region of seeds can be dynamically allocated, more seeds have the opportunity to mutate instead of spending too many mutations on a certain seed. Therefore, seed slicing is proposed.

Fig. 2 shows the process of seed slicing. After a seed has been selected, AMSFuzz first determines which regions have not been sliced and calculates slicing size. Slicing size is shown in Sub Section 3.3.2. AMSFuzz then identifies a slicing region and enters the deterministic stage to mutate this region. Slicing mutation is shown in Sub Section 3.3.3. If the seed has an unmutated region, the seed continues to be sliced when it is selected next time.

3.3.2. Slicing size

The unit of slicing is first determined before assigning a slicing size to a seed. Different mutation strategies perform one mutation with different mutation region sizes in the deterministic stage (e.g., bitflip 1/1 flips 1 bit at a time while bitflip 4/1 flips 4 bits at a time). Since the deterministic stage is to perform all mutations according to seed length, AMSFuzz's slicing mechanism is sliced according to bytes.

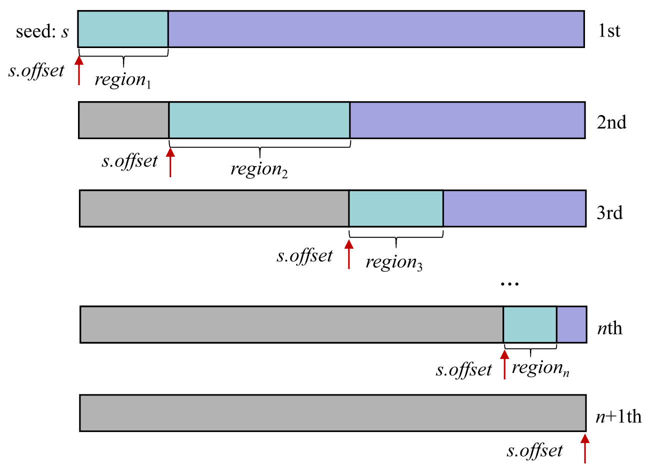

Fig. 2. The process of seed slicing.

AMSFuzz uses the current average number of mutations of discovering a new path to allocate the size of seed slicing during the fuzzing process. The current average number of mutations of discovering a new path reflects average mutations from the start of fuzzing until the determination of seed slicing size. After the program is fuzzed t times, an average number of mutations of generating a new path M_{avg}^t is calculated in the fuzzing process,

$$M_{avg}^t = \begin{cases} \frac{M_{total}^t}{S_{total}^t} & S_{total}^t \neq 0 \\ M_{total}^t & others \end{cases} \tag{5}$$

where M_{total}^t and S_{total}^t indicate a total number of mutations and the number of seeds generated in the seed queue after the program has been fuzzed t times, respectively.

After obtaining an average number of mutations of generating a new path, AMSFuzz first uses it to assign a pre-slice size. AMSFuzz determines the pre-slice size based on the properties of the selected seed. For the first selected seed, AMSFuzz uses the M_{avg}^t to determine the pre-slice size. For seeds selected for slicing again, the pre-slice size of the seed takes into account the result of the last mutation performed by the seed. After the program is fuzzed t times, the pre-slice size $pslice_s^t$ of seed s is calculated as follows,

$$pslice_s^t = \begin{cases} \frac{M_{avg}^t}{n} & fuzz_s = false \\ Min(lslice_s, \alpha \cdot \frac{M_{avg}^t}{n}) & lastf_s = fuzz_s = true \\ Max(\beta \cdot lslice_s, \frac{M_{avg}^t}{n}) & lastf_s = false \& fuzz_s = true \end{cases} \tag{6}$$

where $fuzz_s$ denotes whether the seed s has been selected in the previous fuzzing process. $lastf_s$ denotes whether a new path is found after the seed s is mutated during the last selection, $lslice_s$ denotes the size of the current seed is sliced in the last time, M_{avg}^t denotes the average number of mutations required to generate a seed after the program is fuzzed t times, and n, α, and β indicate coefficients. Here the values are 330, 0.05, and 1.1, respectively.

The region where the seeds have been sliced requires to identify after assigning a pre-slice size. Since the seed length in the queue is different, this makes the pre-slice size ignore the limitation of seed length and unmutated regions. Therefore, we use a seed offset to split seeds into a mutation-complete region and an unmutated region before slicing. The offset $s.offset$ of the seed s is shown in Fig. 2. The mutation-complete region is the head of seeds to seed offset, and the unmutated region is the seed offset to the tail of seeds.

For selected seeds, AMSFuzz first calculates the size of the unmutated region, and if it is larger than the pre-slice size, the seed is sliced according to the pre-slice size. If the size of the unmutated region is less than or equal to the pre-slice size, the remainder of the seed is mutated. After the program is fuzzed t times, the size $slice_s^t$ of seed s is calculated as follows,

$$slice_s^t = \begin{cases} pslice_s^t & pslice_s^t < unmuta_s^t \\ unmuta_s^t & others \end{cases} \tag{7}$$

where $pslice_s^t$ is the pre-assigned slicing size of the seed s when the program has fuzzed t times and $unmuta_s^t$ is the size of the unmutated region of the seed s when the program has fuzzed t times.

3.3.3. Slicing mutation

Algorithm 3: Slicing Mutation

```
Input: Seed inputs: $seeds$, Instrumented target program: $P$
Output: Seed queue: $Q$, Crash set: $C$
1  $Q \leftarrow seeds$
2  $C \leftarrow \emptyset$
3  while TRUE do
4      $s \leftarrow$ ChooseNext($Q$)
5      $s.slice \leftarrow$ AssignSlice($s$)
6      for $op$ in $deterministicOps$ do
7          for $i$ in Range($s.offset, s.offset + s.slice$) do
8              $s' \leftarrow$ Mutation($s, op, i$)
9              $status \leftarrow$ Run_Target($P, s'$)
10             if is_Crash($status$) then
11                 $C \leftarrow C \cup s'$
12                 continue
13             if is_NewCoverage($status$) then
14                 $Q \leftarrow Q \cup s'$
15     $s.offset \leftarrow s.offset + s.slice$
16     if $s.offset = s.length$ then
17         $s.was\_sliced \leftarrow true$
```

AMSFuzz mutates the region of slicing after the slicing size of seeds is allocated. The slicing mutation algorithm of AMSFuzz is shown in Algorithm 3. The seed inputs $seeds$ provided by the user and the instrumented program P are used as inputs of the algorithm. The seed inputs $seeds$ are added to the seed queue Q (Line 1). AMSFuzz enters an infinite loop. A seed s is selected from queue Q (Line 4). The seed is assigned a slicing size (Line 5). For each operator op in the deterministic stage, AMSFuzz determines the region of mutation based on the calculated slicing size and an offset $s.offset$ of seed s, and mutates and executes the program (Lines 8–9). If the seed after mutation causes a crash or produces a new coverage, it is saved in the crash set C and the seed queue Q, respectively (Lines 10–14). The offset $s.offset$ is updated (Line 15). If the offset $s.offset$ points to the end of seed s, the seed completes the slicing process in the deterministic stage and is labeled (Lines 16–17). It is selected again without executing seed slicing.

4. Evaluation

To evaluate the effectiveness of AMSFuzz, a prototype is implemented by extending AFL-2.57b. The experiments are evaluated on a series of real-world programs and LAVA-M dataset. Through these experiments, the following research questions are answered:

RQ1: How does AMSFuzz compare with state-of-the-art fuzzers in terms of path discovery and code coverage?

RQ2: How does AMSFuzz compare with state-of-the-art fuzzers in terms of bug detection?

RQ3: How effective are adaptive operator selection and seed slicing?

RQ4: Can AMSFuzz trigger new bugs?

Table 1
Target programs and their fuzzing configure

Program	Command line	Project version	Format
cxxfilt	c++filt -t	GNU Binutils 2.31	ELF
nm	nm -C @@	GNU Binutils 2.31	ELF
size	size @@	GNU Binutils 2.31	ELF
objdump	objdump -d @@	GNU Binutils 2.31	ELF
strings	strings -a -d -f @@	GNU Binutils 2.31	ELF
tiff2pdf	tiff2pdf @@	libtiff 4.2.0	TIFF
exiv2	exiv2 @@	Exiv2 1.0.0.9	JPG
pngimage	pngimage @@	libpng 1.6.35	PNG
djpeg	djpeg @@	libjpeg-turbo 2.1.1	JPEG
avconv	avconv -i @@ -f null -	libav 12.3	AVI
yara	yara @@	yara 3.6.0	YARA
tcpdump	tcpdump -n -r @@ -v	tcpdump 5.0.0	PCAP

4.1. Experimental setup

Baseline fuzzers. AMSFuzz is compared with the currently popular tools AFL, AFLFast, FairFuzz, and MOPT. The choice of baseline fuzzers is based on the following considerations.

(1) AFL is the current popular mutation-based fuzzing and is widely used in fuzzing.

(2) AFLFast (Böhme et al., 2019) is a variant of AFL that improves path discovery by changing power schedule.

(3) FairFuzz (Lemieux & Sen, 2018) is a tool that improves code coverage by identifying rare branches in programs and improving mutation.

(4) MOPT (Lyu et al., 2019) uses particle swarm optimization (PSO) algorithms to schedule mutation operators in havoc stage.

Dataset. We collected 12 programs for evaluation in Table 1. The selected programs are mainly common test programs. Considering the diversity of programs, the selected programs include well-known development tools (e.g., *nm* (GNU Project, 2021), *cxxfilt* (GNU Project, 2021), *objdump* (GNU Project, 2021), *size* (GNU Project, 2021), and *strings* (GNU Project, 2021)), image processing programs (e.g., *tiff2pdf* (GitLab, 2021), *exiv2* (Exiv2, 2021), *pngimage* (Libpng, 2021), and *djpeg* (GitHub, 2021ab)), audio and video processing tools (eg., *avconv* (GitHub, 2019b)), data processing libraries (eg., *yara* (GitHub, 2021db)), and packet processing programs (eg., *tcpdump* (Tcpdump, 2021)). LAVA-M dataset is also used in the experiment which uses taint analysis to inject errors. Moreover, we randomly download the latest version of open-source projects for testing to discover new bugs.

Initial seeds. For each test program, we randomly collect testcases with valid format as the initial seeds and all fuzzers are configured same seeds (GitHub, 2021ba). In LAVA-M dataset (Dolan-Gavitt et al., 2016), we use seeds provided by the author in the project as initial seeds.

Experiment environment. All experiments are conducted on a server configured with 104 Intel(R) Xeon(R) Platinum 8179M CPU @ 2.40 GHz cores, 32 GB RAM and 64-bit Ubuntu 20.04.1 LTS. We fuzz each program for 24 h and repeat it 5 times to reduce randomness. In all experiments, each fuzzer is single-threaded execution on one CPU core and we leave 4 cores for other processes to keep the workload stable.

4.2. Path discovery & code coverage (RQ1)

The path discovery and code coverage are key metrics in mutation-based fuzzing. The 12 real-world programs in Table 1 are used to evaluate path discovery and code coverage. Path discovery is measured by counting the number of seeds found in the seed queue. Afl-cov (GitHub, 2020a) is used to measure three types of coverage: line, function, and branch. Seeds that are generated by fuzzers are sent to execute the target program to generate gcov code coverage results. Fig. 3 shows average paths found by different fuzzers in real-world programs within 24 h. Table 2 shows the line, function, and

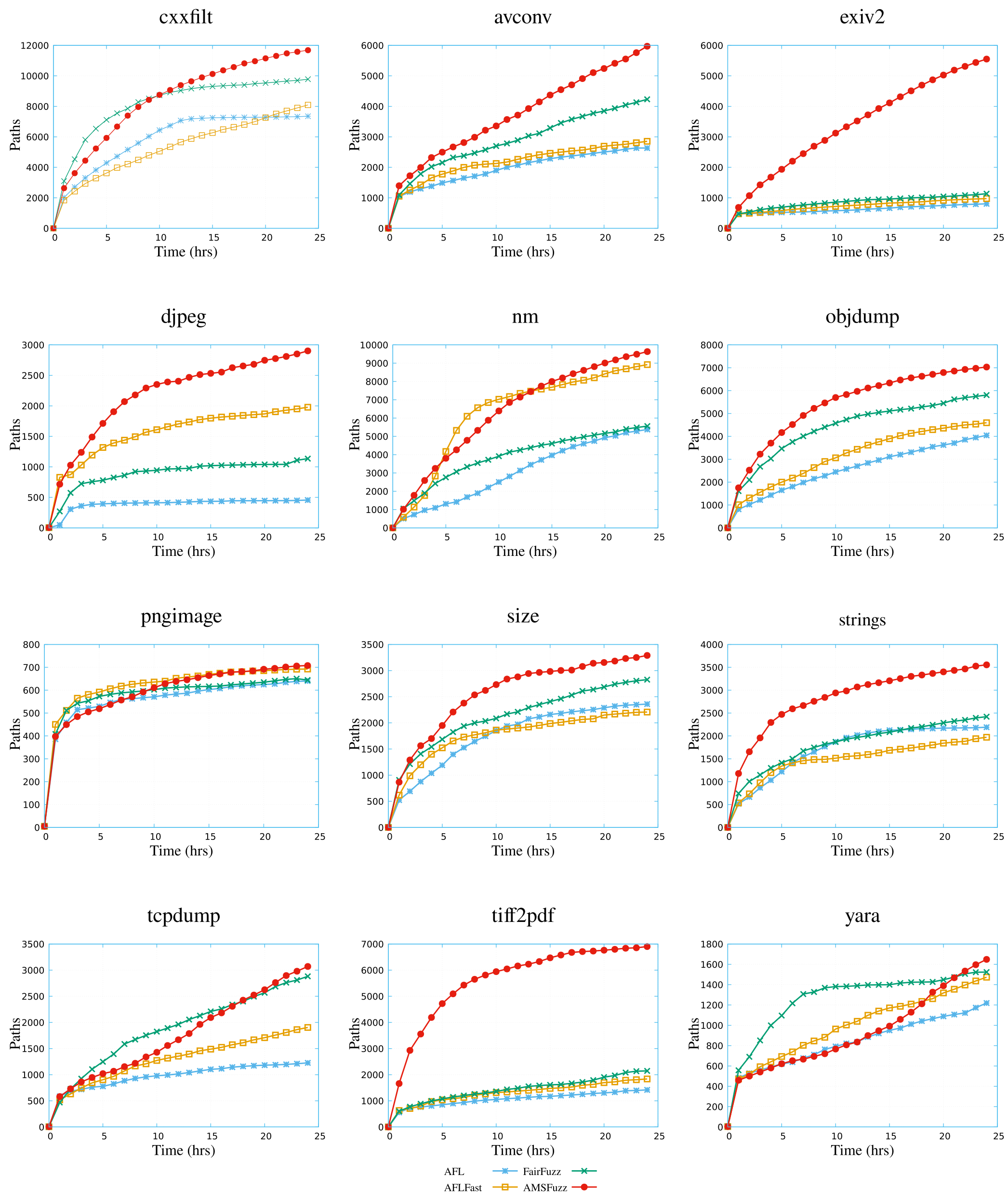

Fig. 3. Average paths found by different fuzzers in real-world programs within 24 h.

branch covered by different fuzzers in various programs. From these experimental results, we can make the following observations.

First, path discovery is restricted by the search space. Most programs such as *cxxfilt*, *avconv*, *tcpdump*, *exiv*2, *nm*, and *yara* have little difference in path discovery within the first hour. However, as time changes, the gap gradually increases in the number of paths. Because the new path space in the program is large when fuzzing starts, and it is easy to mutate to discover new paths. However, with the change of time, the discovered paths increase, and the search space of the path becomes small. This makes it difficult to explore new branches. Therefore, it is critical that seeds use reasonable mutation operators to replace random mutations.

Second, sequential selection of mutation positions affects the discovery of paths. In the experimental program *exiv*2 and *tiff2pdf*, Fig. 3 can clearly show that AMSFuzz is significantly better than other fuzzers. Because other fuzzers spend too much time in the deterministic stage, reducing the speed of path discovery, and thus fall into a deadlock, which also illustrates the necessity of seed slicing.

Third, compared with other fuzzers, AMSFuzz has a better ability to find paths. From the experimental results in Fig. 3, we can clearly observe that except in *yara*, the number of new paths found by AMSFuzz is slightly lower than FairFuzz, but the number of new paths found by AMSFuzz significantly outperforms AFL, AFLFast, and FairFuzz in other programs.

Fourth, as can be observed from Table 2, except in *pngimage*, *yara*, and *tcpdump*, AMSFuzz achieves better line, function, and branch coverage compared with AFL, AFLFast, and FairFuzz. Especially, AMSFuzz achieved 30.5% function coverage in *tiff2pdf*, while AFL only achieved 16.9%. AMSFuzz achieved 14.3% line coverage in *exiv*2, while other fuzzer achieved no more than 10% line coverage. Overall, AMSFuzz achieves better line, function, and branch coverage compared with AFL, AFLFast, and FairFuzz.

4.3. Bug detection (RQ2)

In this subsection, we evaluate our fuzzers in terms of bug detection. LAVA-M dataset and real-world programs are used for experiment

Table 2
Line, function and branch covered by different fuzzers in various programs.

Program	AFL			AFLFast			FairFuzz			AMSFuzz		
	Line	Func	Bran	Line	Func	Bran	Line	Func	Bran	Line	Func	Bran
cxxfilt	2.7%	**3.1%**	**2.4%**	2.7%	**3.1%**	**2.4%**	**2.8%**	**3.1%**	**2.4%**	**2.8%**	**3.1%**	**2.4%**
nm	4.8%	7.1%	4.0%	4.8%	7.0%	4.0%	4.4%	6.8%	3.7%	**5.0%**	**7.4%**	**4.1%**
size	**2.5%**	**4.1%**	**1.9%**	2.1%	3.7%	1.7%	2.3%	3.9%	**1.9%**	**2.5%**	**4.1%**	**1.9%**
objdump	4.3%	6.2%	3.6%	2.8%	4.9%	2.2%	4.6%	6.5%	**3.8%**	**4.7%**	**6.8%**	**3.8%**
strings	2.2%	**3.7%**	**1.7%**	1.8%	3.2%	1.4%	2.0%	3.4%	1.6%	**2.3%**	**3.7%**	**1.7%**
tiff2pdf	11.2%	16.9%	8.4%	11.4%	17.2%	8.5%	10.9%	17.1%	8.2%	**21.0%**	**30.5%**	**17.7%**
exiv2	8.6%	13.1%	3.7%	8.4%	12.7%	3.5%	9.8%	14.3%	4.1%	**14.3%**	**21.2%**	**6.5%**
pngimage	13.2%	19.5%	10.9%	14.0%	19.9%	11.6%	**14.1%**	19.9%	**11.8%**	14.0%	**20.0%**	11.7%
djpeg	13.7%	19.4%	8.8%	17.9%	22.2%	14.4%	16.9%	21.5%	12.2%	**18.2%**	**22.4%**	**14.5%**
avconv	8.9%	11.5%	6.4%	9.0%	11.6%	6.5%	9.3%	11.8%	6.7%	**10.9%**	**13.3%**	**7.9%**
yara	32.0%	34.1%	17.4%	32.7%	34.4%	18.0%	**33.4%**	**34.7%**	**18.6%**	33.2%	34.2%	18.4%
tcpdump	13.1%	25.9%	9.3%	18.1%	31.8%	13.6%	22.3%	**35%**	17.3%	**22.5%**	34.6%	**17.7%**

Table 3
Maximum and average number of bugs triggered by different fuzzers in LAVA-M.

Program	AFL		AFLFast		FairFuzz		AMSFuzz	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max
base64	0	0	0	0	1.2	3	**8.2**	**40**
md5sum	0	0	0	0	0	0	**0.2**	**1**
uniq	0	0	0	0	0	0	**2**	**3**
who	1.8	**3**	1.4	2	1.4	2	**2.4**	**3**
Total	1.8	3	1.4	2	2.6	5	**12.8**	**47**

Table 4
Maximum and average number of bugs triggered by different fuzzers in real-world programs.

Program	AFL		AFLFast		FairFuzz		AMSFuzz	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max
avconv	0	0	0	0	0	0	**42.4**	**127**
nm	0	0	0.4	1	0	0	**5.8**	**20**
cxxfilt	0	0	0.6	3	0	0	**18**	**29**
bento4	59	63	61.6	63	66	109	**157.8**	**171**
jasper	0	0	0	0	**4.4**	**21**	2.2	7
Total	59	63	62.6	67	70.4	130	**226.2**	**354**

Table 5
Average number of paths found by AMS.bandit, MOPT-AFL-ever, AFL-UCB, and AMS-TS in real-world programs within 24 h.

Program	AMS.bandit	MOPT-AFL-ever	AFL-UCB	AMS-TS
cxxfilt	13387	13140	12025	12056
nm	12211	11761	9421	10046
size	3086	3191	2863	3084
objdump	8114	7828	6202	7353
strings	3430	3303	2806	3190
tiff2pdf	6574	7364	5207	6065
exiv2	8926	3946	3628	9208
pngimage	713	672	663	712
djpeg	3487	3613	3291	3420
avconv	12084	9627	7793	12034
yara	2592	2400	2306	2478
tcpdump	7144	5656	6165	7742

evaluation. LAVA-M consists of four test programs: *base*64, *md*5*sum*, *uniq*, and *who*. It uses bug injection techniques to insert bugs into the source code by using taint analysis. It has been used by many fuzzers for evaluation of fuzzers in recent years. The real-world programs are tested using *avconv*, *nm*, *cxxfilt*, *jasper* (GitHub, 2021), and *bento*4 (Bento4, 2021).

Table 3 shows the maximum and average number of bugs triggered by different fuzzers in LAVA-M. Overall, we can see that AMSFuzz triggered the maximum number of bugs in terms of average and maximum number. In average, AFL and AFLFast triggered 0 bugs in *base*64, *md*5*sum*, and *uniq*, while AMSFuzz triggered 8.2, 0.2, and 2 bugs, respectively. In terms of the maximum number, AMSFuzz compared with AFL, AFLFast, and FairFuzz triggered 44, 45, and 42 more bugs in total.

The other evaluation programs are real-world programs. The historical version of target programs is used to evaluate bug detection.

Table 4 shows the maximum and average number of bugs triggered by different fuzzers in real-world programs. Experimentally, AMSFuzz is able to trigger the most bugs on *avconv*, *nm*, *cxxfilt*, *bento*4 compared with AFL, AFLFast, and FairFuzz. Especially on *avconv*, AMSFuzz is able to trigger 127 bugs, other fuzzers have no bug found. FairFuzz triggered the most bugs on the *jasper*, but AMSFuzz triggered the most bugs within 24 h compared with AFL, AFLFast, and FairFuzz from overall.

4.4. The effectiveness of adaptive operator selection and seed slicing (RQ3)

To evaluate the effectiveness of adaptive operator selection and seed slicing, we integrated the approach of adaptive seed selection into AFL as AMS.bandit and the approach of seed slicing into AFL as AMS.slicing, respectively. We also integrated the traditional upper confidence bound (UCB) algorithm into havoc stage of AFL to implement a fuzzer called AFL-UCB and integrated the Thompson sampling (TS) method into the exploitation stage of AMS.bandit to implement the AMS-TS. We compared our two approaches with MOPT, AFL-UCB, AMS-TS, and AFL.

Target programs are fuzzed in AMS.bandit, MOPT-AFL-ever (Lyu et al., 2019), AFL-UCB and AMS-TS. The deterministic stage is skipped in this evaluation. Table 5 shows the average number of paths found by AMS.bandit, MOPT-AFL-ever, AFL-UCB, and AMS-TS in real-world programs within 24 h. To evaluate the effectiveness of seed slicing, we used MOPT, AMS.slicing, and AFL to fuzz target programs. The parameter is set "-L 30" in MOPT. Table 6 shows the average number of paths found by AMS.slicing, MOPT, and AFL in real-world programs within 24 h.

The proposed adaptive operator selection is effective in havoc stage as seen on the experimental results of Table 5. Compared with MOPT-AFL-ever, AMSFuzz.bandit is able to discover more paths in 24 h on programs *cxxfilt*, *nm*, *objdump*, *strings*, *exiv*2, *pngimage*, *avconv*, *yara*, and *tcpdump*. Especially, AMS.bandit found 8926 paths on *exiv*2, while MOPT-AFL-ever found only 3946 paths. Although the number of paths discovered by AMSFuzz.bandit is slightly lower than MOPT-AFL-ever on *size*, *tiff2pdf*, and *yara*, the overall number of paths discovered by AMS.bandit is higher than that of MOPT-AFL-ever. Compared with AFL-UCB, AMS.bandit can find more paths on all target programs. Especially on *exiv*2 and *avconv*, AMS.bandit found 5298 and 4291 more paths, respectively. Compared with AMS-TS, AMS.bandit can discover more paths on programs *cxxfilt*, *nm*, *size*, *objdump*, *strings*, *tiff2pdf*, *pngimage*, *djpeg*, *avconv*, and *yara*.

Our seed slicing mechanism is more effective compared with AFL and MOPT. Table 6 shows that AMS.slicing outperforms MOPT and AFL in path discovery. AMS.slicing discovered the most paths overall compared with MOPT and AFL. We can also see that MOPT discovered lower paths than AFL on *cxxfilt*, while on *pngimage* the opposite is true.

Table 6
Average number of paths found by AMS.slicing, MOPT, and AFL in real-world programs within 24 h.

Program	AMS.slicing	MOPT	AFL
cxxfilt	11342	7041	7353
nm	9401	3955	5374
size	3412	3050	2359
objdump	6786	3891	4044
strings	3445	2720	2190
tiff2pdf	6836	5763	1418
exiv2	5678	2753	803
pngimage	770	803	640
djpeg	3323	2638	455
avconv	6924	5712	2633
yara	2362	1545	1219
tcpdump	4409	4616	1226

This is because the seeds have different lengths and MOPT uses a set time to improve path discovery. However, the time setting cannot be dynamically corrected during fuzzing process, whereas our seed slicing avoids this problem.

4.5. New bugs (RQ4)

To further evaluate AMSFuzz, we used the designed tool to fuzz the latest version of programs. We chose 5 open-source projects to test, including GPAC (GitHub, 2021ca), Vim (GitHub, 2021cb), Binaryen (GitHub, 2021aa), HDF5 (GitHub, 2021da), and DjVuLibre (SourceForge, 2021). Among them, GPAC is an open-source multimedia framework. Vim is a common text editor which has been stared 25.9k times on GitHub. Binaryen is a compiler and toolchain infrastructure library for WebAssembly. HDF5 is a high performance data software library, and DjVuLibre is an open-source DjVu library and viewer. They are widely active in the open-source community. In our experiments, AddressSanitizer (LLVM, 2021) is used to compile and link programs to find more bugs. For each program, we use AMSFuzz fuzz for 24 h. We also screen collected bugs to reduce repeatability.

We disclosed issues in projects and reported them publicly to maintainers. Some of the bug have been fixed and CVE IDs have been awarded. Table 7 shows new bugs found by AMSFuzz in open-source projects. AMSFuzz found a total of 17 new bugs, 15 of which obtained CVE IDs. The remaining two bugs are currently awaiting maintainers fixes and an official response. In addition, AMSFuzz found many different types of bugs including untrusted pointer dereference, heap-buffer-overflow, stack-buffer-overflow, divide by zero, infinite loop, segmentation fault, and null pointer dereference.

5. Discussion

Fuzzing is a practical and automated software testing technique. The current popular mutation-based fuzzer, AFL ignores the impact of seed length and schedule of mutation operators, which results in a large number of ineffective mutation operators on the seed. AMSFuzz improves the efficiency of fuzzing by extending AFL. Experimental results show that AMSFuzz is effective in path discovery and bug detection compared with AFL, AFLFast, FairFuzz, and MOPT.

Different from other works (Bali et al., 2014; Cui et al., 2018), the adaptive operator selection implemented in AMSFuzz mainly models fuzzing as a MAB model, while MOPT uses PSO schedule mutation operators. Seed slicing in AMSFuzz focuses on the selection of mutation positions in the deterministic stage and it gives multiple seeds the opportunity to mutate in a certain time. MOPT disables the deterministic stage when no new path is found within the set time. The fixed time can be set by users, but it is a problem that how to set the time for different programs and this may miss some critical paths in the unmutated region of the seed.

We model fuzzing as a MAB model, and propose an adaptive mutation schedule method in havoc stage. Like traditional MAB methods, our method constantly selects actions to obtain the maximum rewards as much as possible. Different from the traditional MAB methods such as UCB (Auer & Ortner, 2010) and TS (Chapelle & Li, 2011), we customized the exploration and exploitation stages of MAB model according to the process of fuzzing. We adopt random mutation operator selection in the exploration stage to obtain a more accurate probability and the mutation operator selection based on probability distribution in the exploitation stage to obtain the more rewards as much as possible. Although our method is similar to TS in terms of using probability distributions, we consider the restriction of the path search space and the proposed the operator selection based on probability distributions is performed only in a customized exploitation stage, rather than occurring throughout the fuzzing process. In addition, unlike the traditional TS, our proposed method has only one probability distribution. In contrast, the TS has one probability distribution for each arm.

AMSFuzz is implemented based on a popular tool, AFL. The approach proposed in this paper can be integrated into other variants of AFL to improve the efficiency of the fuzzer, which is faster than other fuzzers using taint analysis (Bekrar et al., 2012). However, there is still great scope for AMSFuzz to address magic bytes in programs, and future works could integrate symbolic execution into AMSFuzz to address these issues.

6. Conclusion and future work

In this paper, we present two techniques, an adaptive mutation schedule and a seed slicing mechanism to schedule mutation operators and mutation regions. We implement a prototype, AMSFuzz by extending and enhancing the popular fuzzer AFL, and our evaluation demonstrates that AMSFuzz outperforms state-of-the art fuzzers, such as AFL, AFLFast, FairFuzz, and MOPT in path discovery, coverage, and bug detection. AMSFuzz also found 17 new bugs, 15 of which were revealed and received CVE IDs.

There is still some valuable works to be studied in the field of fuzzing. An effective fuzzer should be applied to specific applications and solve specific problems. Our future works focus on two issues. On the one hand, fuzzing is used in specific applications, such as network protocols. We are customizing application-specific fuzzers. On the other hand, we continue to improve AMSFuzz and integrate other techniques, such as symbolic execution and taint analysis, to solve the magic bytes problem and branch constraint problem. We also use fuzzing techniques and vulnerability features to implement vulnerability-oriented fuzzers.

CRediT authorship contribution statement

Xiaoqi Zhao: Conceptualization, Methodology, Software, Validation,Writing – original draft. **Haipeng Qu:** Conceptualization, Methodology, Funding acquisition, Formal analysis. **Jianliang Xu:** Resources, Project administration, Supervision. **Shuo Li:** Software, Data curation, Investigation, Visualization. **Gai-Ge Wang:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work was supported by the National Natural Science Foundation of China [grant number 61827810].

Table 7
New bugs found by AMSFuzz in open-source projects.

Project version	Command	Type	CVE ID&Issue
Vim 8.2	vim -u NONE -X -Z -e -s -S @@ -c :qa!	heap-based buffer overflow	CVE-2022-0261 (CVE, 2022o)
HDF5 1.13.1–1	h5format_convert -n @@	heap-buffer-overflow	CVE-2021-45832 (CVE, 2022h)
HDF5 1.13.1–1	h5dump @@	stack-buffer-overflow	CVE-2021-45833 (CVE, 2022i)
HDF5 1.13.1–1	h5format_convert -n @@	heap-buffer-overflow	CVE-2021-45830 (CVE, 2022f)
Binaryen 103	wasm-opt @@	assertion failure	CVE-2021-45290 (CVE, 2022a)
Binaryen 103	wasm-dis @@	untrusted pointer dereference	CVE-2021-45293 (CVE, 2022d)
Binaryen 103	wasm-ctor-eval @@	stack overflow	CVE-2021-46050 (CVE, 2022n)
GPAC 1.1.0	MP4Box -lsr @@	segmentation fault	CVE-2021-45291 (CVE, 2022b)
GPAC 1.1.0	MP4Box -hint @@	infinite loop	CVE-2021-45297 (CVE, 2022e)
GPAC 1.1.0	MP4Box -bt @@	null pointer dereference	CVE-2021-45831 (CVE, 2022g)
GPAC 1.1.0	MP4Box -hint @@	segmentation fault	CVE-2021-46041 (CVE, 2022k)
GPAC 1.1.0	MP4Box -hint @@	null pointer dereference	CVE-2021-46038 (CVE, 2022ja)
GPAC 1.1.0	MP4Box -hint @@	null pointer dereference	CVE-2021-46039 (CVE, 2022jb)
GPAC 1.1.0	MP4Box -hint @@	null pointer dereference	CVE-2021-46043 (CVE, 2022l)
GPAC 1.1.0	MP4Box -par 1=4:3 @@	null pointer dereference	CVE-2021-46049 (CVE, 2022m)
DjVuLibre 3.5.28	c44 @@	divide by zero	issue344 (SourceForge, 2022a)
DjVuLibre 3.5.28	djvups @@	divide by zero	issue345 (SourceForge, 2022b)

References

Adobe (2009). Adobe reader and acrobat security initiative. Retrieved from https://blogs.adobe.com/security/2009/05/adobe_reader_and_acrobat_secur.html. (Accessed 30 July 2021).

Auer, P., & Ortner, R. (2010). UCB revisited: Improved regret bounds for the stochastic multi-armed bandit problem. *Periodica Mathematica Hungarica*, *61*(1–2), 55–65. http://dx.doi.org/10.1007/s10998-010-3055-6.

Avner, O., & Mannor, S. (2019). Multi-user communication networks: A coordinated multi-armed bandit approach. *IEEE/ACM Transactions on Networking*, *27*(6), 2192–2207. http://dx.doi.org/10.1109/tnet.2019.2935043.

Bali, S., Jha, P., Kumar, U. D., & Pham, H. (2014). Fuzzy multi-objective build-or-buy approach for component selection of fault tolerant software system under consensus recovery block scheme with mandatory redundancy in critical modules. *International Journal of Artificial Intelligence and Soft Computing*, *4*(2–3), 98–119. http://dx.doi.org/10.1504/IJAISC.2014.062815.

Bekrar, S., Bekrar, C., Groz, R., & Mounier, L. (2012). A taint based approach for smart fuzzing. In *2012 IEEE Fifth international conference on software testing, verification and validation* (pp. 818–825). http://dx.doi.org/10.1109/icst.2012.182.

Bento4 (2021). Bento4. Retrieved from http://www.bento4.com. (Accessed 30 May 2021).

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). http://dx.doi.org/10.1145/3133956.3134020.

Böhme, M., Pham, V., & Roychoudhury, A. (2019). Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, *45*(5), 489–506. http://dx.doi.org/10.1109/tse.2017.2785841.

Chapelle, O., & Li, L. (2011). An empirical evaluation of thompson sampling. *Advances in Neural Information Processing Systems, 24.*

Cheng, W., Wang, H., Li, Y., Qin, S., Liu, Y., Xu, Z., Chen, H., Xie, X., Pu, G., & Liu, T. (2020). MemLock: Memory usage guided fuzzing. In *42nd International conference on software engineering* (pp. 765–777). http://dx.doi.org/10.1145/3377811.3380396.

Chris, E., Matt, M., & Tavis, O. (2011). Fuzzing at scale. Retrieved from https://security.googleblog.com/2011/08/fuzzing-at-scale.html. (Accessed 30 July 2021).

Cuevas, E., Echavarría, A., & Ramírez-Ortegón, M. A. (2014). An optimization algorithm inspired by the states of matter that improves the balance between exploration and exploitation. *Applied Intelligence: The International Journal of Artificial Intelligence, Neural Networks, and Complex Problem-Solving Technologies*, *40*(2), 256–272. http://dx.doi.org/10.1007/s10489-013-0458-0.

Cui, Z., Xue, F., Cai, X., Cao, Y., Wang, G.-g., & Chen, J. (2018). Detection of malicious code variants based on deep learning. *IEEE Transactions on Industrial Informatics*, *14*(7), 3187–3196. http://dx.doi.org/10.1109/tii.2018.2822680.

CVE (2022a). CVE-2021-45290. Retrieved from https://www.cvedetails.com/cve/CVE-2021-45290. (Accessed 10 May 2022).

CVE (2022b). CVE-2021-45291. Retrieved from https://www.cvedetails.com/cve/CVE-2021-45291. (Accessed 10 May 2022).

CVE (2022d). CVE-2021-45293. Retrieved from https://www.cvedetails.com/cve/CVE-2021-45293. (Accessed 10 May 2022).

CVE (2022e). CVE-2021-45297. Retrieved from https://www.cvedetails.com/cve/CVE-2021-45297. (Accessed 10 May 2022).

CVE (2022f). CVE-2021-45830. Retrieved from https://www.cvedetails.com/cve/CVE-2021-45830. (Accessed 10 May 2022).

CVE (2022g). CVE-2021-45831. Retrieved from https://www.cvedetails.com/cve/CVE-2021-45831. (Accessed 10 May 2022).

CVE (2022h). CVE-2021-45832. Retrieved from https://www.cvedetails.com/cve/CVE-2021-45832. (Accessed 10 May 2022).

CVE (2022i). CVE-2021-45833. Retrieved from https://www.cvedetails.com/cve/CVE-2021-45833. (Accessed 10 May 2022).

CVE (2022j). CVE-2021-46038. Retrieved from https://www.cvedetails.com/cve/CVE-2021-46038. (Accessed 10 May 2022).

CVE (2022j). CVE-2021-46039. Retrieved from https://www.cvedetails.com/cve/CVE-2021-46039. (Accessed 10 May 2022).

CVE (2022k). CVE-2021-46041. Retrieved from https://www.cvedetails.com/cve/CVE-2021-46041. (Accessed 10 May 2022).

CVE (2022l). CVE-2021-46043. Retrieved from https://www.cvedetails.com/cve/CVE-2021-46043. (Accessed 10 May 2022).

CVE (2022m). CVE-2021-46049. Retrieved from https://www.cvedetails.com/cve/CVE-2021-46049. (Accessed 10 May 2022).

CVE (2022n). CVE-2021-46050. Retrieved from https://www.cvedetails.com/cve/CVE-2021-46050. (Accessed 10 May 2022).

CVE (2022o). Vim. Retrieved from https://www.cvedetails.com/cve/CVE-2022-0261/?q=cve-2022-0261 Accessed May 10, 2022.

Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., Ulrich, F., & Whelan, R. (2016). LAVA: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on security and privacy* (pp. 110–121). http://dx.doi.org/10.1109/sp.2016.15.

Duan, H., Zhao, W., Wang, G., & Feng, X. (2012). Test-sheet composition using analytic hierarchy process and hybrid metaheuristic algorithm TS/BBO. *Mathematical Problems in Engineering*, *2012*, http://dx.doi.org/10.1155/2012/712752.

Exiv2 (2021). Exiv2. Retrieved from https://exiv2.org/. (Accessed 23 July 2021).

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* (pp. 679–696). http://dx.doi.org/10.1109/sp.2018.00040.

GitHub (2019a). American fuzzy lop. Retrieved from https://github.com/google/AFL. (Accessed 1 March 2021).

GitHub (2019b). Libav. Retrieved from https://github.com/libav/libav/. (Accessed 23 July 2021).

GitHub (2020a). Afl-cov. Retrieved from https://github.com/soh0ro0t/afl-cov. (Accessed 23 September 2021).

GitHub (2020b). Onefuzz. Retrieved from https://github.com/microsoft/onefuzz. (Accessed 23 July 2021).

GitHub (2021). Jasper. Retrieved from https://github.com/mdadams/jasper. (Accessed 30 May 2021).

GitHub (2021a). Binaryen. Retrieved from https://github.com/WebAssembly/binaryen. (Accessed 10 December 2021).

GitHub (2021a). Libjpeg-turbo. Retrieved from https://github.com/libjpeg-turbo/libjpeg-turbo. (Accessed 23 July 2021).

GitHub (2021b). Fuzzdata. Retrieved from https://github.com/MozillaSecurity/fuzzdata.git. (Accessed 30 March 2021).

GitHub (2021b). Trinity. Retrieved from https://github.com/kernelslacker/trinity. (Accessed 30 July 2021).

GitHub (2021c). GPAC. Retrieved from https://github.com/gpac/gpac. (Accessed 10 December 2021).

GitHub (2021c). Vim. Retrieved from https://github.com/vim/vim. (Accessed 20 December 2021).

GitHub (2021d). HDF5. Retrieved from https://github.com/HDFGroup/hdf5. (Accessed 10 December 2021).

GitHub (2021d). Yara. Retrieved from https://github.com/VirusTotal/yara. (Accessed 23 July 2021).

GitLab (2021). Libtiff. Retrieved from https://gitlab.com/libtiff/libtiff. (Accessed 23 July 2021).

GNU Project (2021). GNU binutils. Retrieved from https://www.gnu.org/software/binutils/ (Accessed 23 July 2021).

Godefroid, P., Peleg, H., & Singh, R. (2017). Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International conference on automated software engineering* (pp. 50–59). http://dx.doi.org/10.1109/ase.2017.8115618.

He, J., Balunović, M., Ambroladze, N., Tsankov, P., & Vechev, M. (2019). Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on computer and communications security* (pp. 531–548). http://dx.doi.org/10.1145/3319535.3363230.

Kim, K., Jeong, D. R., Kim, C. H., Jang, Y., Shin, I., & Lee, B. (2020). HFL: Hybrid fuzzing on the linux kernel. In *Proceedings of the 2020 Annual network and distributed system security symposium*. http://dx.doi.org/10.14722/ndss.2020.24018.

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). http://dx.doi.org/10.1145/3238147.3238176.

Li, X., Sun, L., Qu, H., Jang, R., & Yan, Z. (2021). OTA: An operation-oriented time allocation strategy for greybox fuzzing. In *28th IEEE International conference onsoftware analysis, evolution and reengineering* (pp. 108–118). http://dx.doi.org/10.1109/saner50967.2021.00019.

Li, Y., Xue, Y., Chen, H., Wu, X., Zhang, C., Xie, X., Wang, H., & Liu, Y. (2019). Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering* (pp. 533–544). http://dx.doi.org/10.1145/3338906.3338975.

Li, F., Yu, D., Yang, H., Yu, J., Karl, H., & Cheng, X. (2020). Multi-armed-bandit-based spectrum scheduling algorithms in wireless networks: A survey. *IEEE Wireless Communications*, *27*(1), 24–30. http://dx.doi.org/10.1109/mwc.001.1900280.

Libpng (2021). Libpng. Retrieved from http://www.libpng.org/pub/png/libpng.html. (Accessed 23 July 2021).

LLVM (2021). Addresssanitizer. Retrieved from https://clang.llvm.org/docs/AddressSanitizer.html Accessed July 23, 2021.

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* (pp. 1949–1966).

Mahajan, A., & Teneketzis, D. (2008). Multi-armed bandit problems. In *Foundations and applications of sensor management* (pp. 121–151).

Mansur, M. N., Christakis, M., Wüstholz, V., & Zhang, F. (2020). Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In *Proceedings of the 28th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering* (pp. 701–712). http://dx.doi.org/10.1145/3368089.3409763.

Miller, B. P., Fredriksen, L., & So, B. (1990). An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, *33*, 32–44. http://dx.doi.org/10.1145/96267.96279.

Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., & Bos, H. (2017). Vuzzer: Application-aware evolutionary fuzzing. In *24th Annual network and distributed system security symposium* (pp. 1–14). http://dx.doi.org/10.14722/ndss.2017.23404.

Rizk-Allah, R. M., El-Sehiemy, R. A., Deb, S., & Wang, G.-G. (2017). A novel fruit fly framework for multi-objective shape design of tubular linear synchronous motor. *The Journal of Supercomputing*, *73*(3), 1235–1256. http://dx.doi.org/10.1007/s11227-016-1806-8.

Schumilo, S., Aschermann, C., Abbasi, A., Worner, S., & Holz, T. (2020). HYPER-CUBE: High-dimensional hypervisor fuzzing. In *27th Annual network and distributed system security symposium* (pp. 23–26). http://dx.doi.org/10.14722/ndss.2020.23096.

Schumilo, S., Aschermann, C., Abbasi, A., Wörner, S., & Holz, T. (2021). Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th USENIX security symposium* (pp. 2597–2614).

Scott, S. L. (2015). Multi-armed bandit experiments in the online service economy. *Applied Stochastic Models in Business and Industry*, *31*(1), 37–45. http://dx.doi.org/10.1002/asmb.2107.

Serebryany, K. (2016). Continuous fuzzing with libfuzzer and AddressSanitizer. In *2016 IEEE Cybersecurity development* (pp. 157–157). http://dx.doi.org/10.1109/secdev.2016.043.

SourceForge (2021). DjVuLibre. Retrieved from https://sourceforge.net/projects/djvu/. (Accessed 10 December 2021).

SourceForge (2022). Issue344. Retrieved from https://sourceforge.net/p/djvu/bugs/344. (Accessed 10 May 2022).

SourceForge (2022). Issue345. Retrieved from https://sourceforge.net/p/djvu/bugs/345 (Accessed 10 May 2022).

Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., & Vigna, G. (2016). Driller: Augmenting fuzzing through selective symbolic execution. In *23rd Annual network and distributed system security symposium* (pp. 1–16). http://dx.doi.org/10.14722/ndss.2016.23368.

Sun, L., Li, X., Qu, H., & Zhang, X. (2020). AFLTurbo: Speed up path discovery for greybox fuzzing. In *2020 IEEE 31st International symposium on software reliability engineering* (pp. 81–91). http://dx.doi.org/10.1109/issre5003.2020.00017.

Tcpdump (2021). Tcpdump. Retrieved from http://www.tcpdump.org. (Accessed 23 July 2021).

Villar, S. S., Bowden, J., & Wason, J. (2015). Multi-armed bandit models for the optimal design of clinical trials: benefits and challenges. *Statistical Science: A Review Journal of the Institute of Mathematical Statistics*, *30*(2), 199. http://dx.doi.org/10.1214/14-sts504.

Wang, G.-G., Cai, X., Cui, Z., Min, G., & Chen, J. (2020). High performance computing for cyber physical social systems by using evolutionary multi-objective optimization algorithm. *IEEE Transactions on Emerging Topics in Computing*, *8*(1), 20–30. http://dx.doi.org/10.1109/tetc.2017.2703784.

Wang, J., Chen, B., Wei, L., & Liu, Y. (2019). Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International conference on software engineering* (pp. 724–735). http://dx.doi.org/10.1109/icse.2019.00081.

Wang, G.-G., Gao, D., & Pedrycz, W. (2022). Solving multi-objective fuzzy job-shop scheduling problem by a hybrid adaptive differential evolution algorithm. *IEEE Transactions on Industrial Informatics*, http://dx.doi.org/10.1109/tii.2022.3165636.

Wang, F., & Shoshitaishvili, Y. (2017). Angr-the next generation of binary analysis. In *2017 IEEE cybersecurity development* (pp. 8–9). http://dx.doi.org/10.1109/SecDev.2017.14.

Wang, G.-G., & Tan, Y. (2019). Improving metaheuristic algorithms with information feedback models. *IEEE Transactions on Cybernetics*, *49*(2), 542–555. http://dx.doi.org/10.1109/tcyb.2017.2780274.

Wang, H., Xie, X., Li, Y., Wen, C., Li, Y., Liu, Y., Qin, S., Chen, H., & Sui, Y. (2020). Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *42nd International conference on software engineering* (pp. 999–1010). http://dx.doi.org/10.1145/3377811.3380386.

Winterer, D., Zhang, C., & Su, Z. (2020). Validating SMT solvers via semantic fusion. In *Proceedings of the 41st ACM SIGPLAN Conference on programming language design and implementation* (pp. 718–730). http://dx.doi.org/10.1145/3385412.3385985.

Xie, X., Ma, L., Juefei-Xu, F., Xue, M., Chen, H., Liu, Y., Zhao, J., Li, B., Yin, J., & See, S. (2019). DeepHunter: A coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International symposium on software testing and analysis* (pp. 146–157). http://dx.doi.org/10.1021/acs.jcim.8b00542.s002.

You, W., Liu, X., Ma, S., Perry, D., Zhang, X., & Liang, B. (2019). SLF: Fuzzing without valid seed inputs. In *2019 IEEE/ACM 41st International conference on software engineering* (pp. 712–723). http://dx.doi.org/10.1109/icse.2019.00080.

Yu, B., Wang, P., Yue, T., & Tang, Y. (2019). Poster: Fuzzing iot firmware via multi-stage message generation. In *Proceedings of the 2019 ACM SIGSAC Conference on computer and communications security* (pp. 2525–2527). http://dx.doi.org/10.1145/3319535.3363247.

Yue, T., Wang, P., Tang, Y., Wang, E., Yu, B., Lu, K., & Zhou, X. (2020). EcoFuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *29th USENIX security symposium* (pp. 2307–2324).

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* (pp. 745–761).

Zhao, X., Qu, H., Lv, W., Li, S., & Xu, J. (2021). MooFuzz: Many-objective optimization seed schedule for fuzzer. *Mathematics*, *9*(3), http://dx.doi.org/10.3390/math9030205.