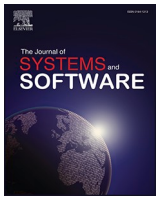
The Journal of Systems and Software 208 (2024) 111880

Contents lists available at ScienceDirect

The Journal of Systems & Software

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



SYNTONY: Potential-aware fuzzing with particle swarm optimization

Xiajing Wang [a], Rui Ma [b,*], Wei Huo [c], Zheng Zhang [b], Jinyuan He [b], Chaonan Zhang [a], Donghai Tian [b]

[a] The Open University of China, Beijing, China
[b] School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China
[c] Institute of Information Engineering, Chinese Academy of Sciences, School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China

ARTICLE INFO

Keywords:
Particle swarm optimization (PSO)
Coverage-guided fuzzing
AFL
Seed selection
Multiple criteria

ABSTRACT

Fuzzing has gained significant traction in academic research as well as industry thanks to its effectiveness for discovering software vulnerabilities. However, even the state-of-the-art fuzzers are not very efficient at identifying promising seeds. Coverage-guided fuzzers, while fast and scalable, usually employ single criterion to evaluate the quality of seeds, which may incur bias and pass up optimal seeds. In this paper, we devise a novel potential-aware fuzzing scheme, namely SYNTONY, which seeks to measure seed potential utilizing multiple objectives and prioritize promising seeds that are more likely to generate interesting seeds via mutation. More specifically, SYNTONY leverages efficient swarm intelligence techniques like Particle Swarm Optimization (PSO) to explore multi-criteria seed selection, which allows SYNTONY to choose effectively promising seeds. Furthermore, we introduce decent power scheduling strategy to discover significantly more paths or crashes by gravitating towards more potential seeds. We implement this scheme on top of several state-of-the-art fuzzers, i.e., AFL, AFL++, FairFuzz, and PTFuzz. Our evaluations on 11 popular real-world programs demonstrate that SYNTONY significantly increases the number of unique crashes triggered and edge coverage discovered by 132.06 % and 28.69 % over AFL++. Further comparison also shows that SYNTONY outperforms other state-of-the-art fuzzers, e.g., AFL, FairFuzz, and PTFuzz. Also, extensive evaluations illustrate that SYNTONY provides a great compatibility and expansibility, while introducing negligible overhead.

1. Introduction

Software vulnerabilities remain one of the most significant threats facing cyber-security. Fuzzing, an automated software testing technique, is widely regarded as a valuable vulnerability hunting method due to its speed, simplicity, and effectiveness (Serebryany, 2016; Swiecki, 2019). At a high level, fuzzing refers to a process of repetitively executing target program with modified or fuzzed test inputs to potentially trigger vulnerabilities. Most notably, coverage-guided fuzzing becomes highly popular especially since AFL (Zalewski, 2019) has shown its vast amount of empirical evidence in discovering numerous real-world software vulnerabilities. In general, AFL adopts an evolving algorithm to drive it towards a higher code coverage.

Recently, a large body of efficient fuzzers were developed based on AFL. More specifically, multiple approaches that typically make use of state-of-the-art program analysis techniques, such as taint tracking (Chen and Chen, 2018; Jain et al., 2018; Gan et al., 2020) or symbolic execution (Stephens et al., 2016; Yun et al., 2018; Zhao et al., 2019), were proposed to boost the seed generation (Wang et al., 2017a; Godefroid et al., 2017; Aschermann et al., 2019a), seed selection (Böhme et al., 2016; Rawat et al., 2017; Wang et al., 2016), seed mutation (Rajpal et al., 2017; Karamcheti et al., 2018; Böttinger et al., 2018; Lyu et al., 2019), and the performance of fuzzing (Zhang et al., 2018; Xu et al., 2017; Chen et al., 2019a; Nagy and Hicks, 2019). While great progress has been achieved in the field of fuzzing, three critical questions regarding seed potential remain to be addressed.

First, *how to measure seeds potential*? Coverage-guided fuzzers (e.g., AFL and its descendants) usually predefine a measurement criterion to characterize which seeds are high-quality. For instance, AFL primarily employs two types of criteria, i.e., execution time or seed size, to estimate which seeds are promising and which seeds should be selected for next fuzzing. AFLFast (Böhme et al., 2016) believes that seeds exercising low-frequency paths are promising. Although their success, using only one or two single criteria is likely to cause deviations and miss promising

* Corresponding author.
E-mail address: mary@bit.edu.cn (R. Ma).

https://doi.org/10.1016/j.jss.2023.111880
Received 15 November 2022; Received in revised form 11 October 2023; Accepted 14 October 2023
Available online 16 October 2023

seeds (Li et al., 2019a). Thus, a more all-sided measurement criterion for seed potential is crucial to guide seed selection and power scheduling in fuzzing.

Second, *how to efficiently select promising seed for fuzzing*? Apparently, a decent seed selection strategy plays a vital role in improving the efficiency of fuzzing. Prior projects (Rawat et al., 2017; Wang et al., 2016), for example, leverage heavyweight program analysis techniques like taint analysis, symbolic execution, etc., to seek most promising seeds for mutating, which may hamper their scalability. Cerebro (Li et al., 2019a) utilizes the Pareto frontier and non-dominated sorting to search for the next seed, but its Pareto frontier calculation and convergence process will bring a certain performance overhead. Consequently, a lightweight yet high-efficiency seed selection strategy without complex program analysis or additional performance overhead is essential for target binaries.

Third, *how to allocate proper energy for selected seed*? Particularly, fuzzers like AFL often endow each seed with an energy that specifies the number of new seeds to be mutated from that seed (i.e., power scheduling). To maximize benefits, fuzzers should ideally assign more mutation energy to promising seeds. Recent studies (Böhme et al., 2017; Chen et al., 2018) optimize power scheduling through allocating more energy to seeds that are closer to the target sites. Nevertheless, less attention has been paid to how to effectively impart proper energy to seeds using seed potential guided by coverage feedback. Therefore, a decent power scheduling is indispensable to boost the efficiency of coverage-guided fuzzers.

Our approach. In this paper, we proposed a potential-aware lightweight fuzzing scheme SYNTONY, which tries to optimize seed selection and power scheduling without any heavyweight program analysis techniques, to address the aforementioned questions. More specifically, SYNTONY first collects multiple criteria of seeds provided by executing target program to characterize seed potential. This integrated seed potential allows SYNTONY to efficiently perform potential-aware seed selection as well as power scheduling. Then, SYNTONY employs a lightweight Particle Swarm Optimization (i.e., PSO) algorithm to achieve multi-criteria seed selection strategy on the basis of seed potential, unlike prior fuzzers that choose seeds in a single criterion. By doing this, SYNTONY could select the most promising next seed for fuzzing. Also, SYNTONY leverages diverse seed potential to facilitate power scheduling. Specifically, it uses path prospect and crash prospect to allocates more energy to seeds that are more likely to discover a crash or new path in the future. Such potential-aware strategy provides an excellent chance for fuzzing, in which the fuzzer can incessantly select promising seed for mutation and allocate more mutation energy to that seed.

We implemented our scheme, SYNTONY, on top of AFL (Zalewski, 2019) and AFL++ (Zalewski, 2020). Notably, our fuzzer preserves the ease-of-use of AFL or AFL++, and the seed selection strategy as well as power scheduling strategy proposed by SYNTONY are orthogonal to other mainstream approaches that manage to improve the performance of baselines by optimizing seed mutation strategy, by leveraging hardware mechanism, or by helping baselines pass magic bytes checks. Hence, SYNTONY is easy to integrate with a wide range of existing fuzzers to simultaneously increase code coverage and the number of bugs or crashes discovered. We have applied SYNTONY to other state-of-the-art fuzzers, i.e., FairFuzz (Lemieux and Sen, 2017) and PTFuzz[1] (Zhang et al., 2018), and implemented SYNTONY-FairFuzz as well as SYNTONY-PTFuzz, respectively.

Furthermore, we evaluated these prototypes on LAVA-M dataset (Dolan-Gavitt et al., 2016) and several real-world programs to examine the relative effectiveness of SYNTONY against six single selection criteria, original AFL++, and other advanced fuzzers. The results are encouraging. SYNTONY explored around 28.69 % more edges and 25.02 % more unique paths than AFL++ in all programs. Further, SYNTONY discovered significantly more unique crashes than AFL++ in 9 out of 11 real-world programs. In total, SYNTONY discovered 304 unique crashes in these programs, whereas AFL++ found 131 unique crashes. Also, SYNTONY-AFL, SYNTONY-FairFuzz, and SYNTONY-PTFuzz significantly outperform their baseline fuzzers (i.e., found 1.89 × , 3 × , and 1.65 × more crashes than AFL, FairFuzz, and PTFuzz). We further demonstrated the rationality and low overhead of SYNTONY.

Contributions. In summary, this paper makes the following contributions.

- *Seed potential with multi-criteria.* We employ multiple native or auxiliary criteria of seeds to measure seed potential from diverse perspectives, which is used to guide seed selection and power scheduling in fuzzing.
- *Potential-aware selection strategies.* We provide a potential-aware seed selection strategy that utilizes a lightweight PSO algorithm and seed potential for prioritizing the most potential seed, thereby making our scheme more efficient.
- *Efficient power scheduling.* We devise an efficient power scheduling strategy to control the number of seeds generated from a specific seed, with the objective to allocate more energy to potential seeds.
- *Tool.* We design, implement, and evaluate our technique as part of SYNTONY and demonstrate that it discovers, on average, 132.06 % more unique crashes and 28.69 % more edges than AFL++. We further demonstrate the compatibility and performance of SYNTONY.

The remainder of this paper is organized as follows. Section 2 clarifies the background and analyzes the motivation by a simple experiment. Section 3 and Section 4 depict the design and implementation of SYNTONY, respectively. In Section 5, we describe various experiments conducted for evaluating the unique paths and crashes of SYNTONY against excellent fuzzers on real-world programs and LAVA-M. Section 6 discusses SYNTONY's limitations and future work. Section 7 presents the related works, and Section 8 offers the conclusion.

2. Preliminaries and motivation

In this section, we provide background on coverage-guided fuzzing and relevant Particle Swarm Optimization (PSO) algorithm. Additionally, we illustrate the challenge of current fuzzing techniques by means of a motivating experiment.

2.1. Coverage-guided fuzzing

In the past few years, coverage-guided fuzzing (Serebryany, 2016; Swiecki, 2019; Zalewski, 2019; Chen and Chen, 2018; Böhme et al., 2016; Gan et al., 2018) has been an active field of research. Normally, they are based on the observation that increasing code coverage could bring about better crash detection (Lemieux and Sen, 2017). Coverage-guided fuzzing thus aims to explore as many program paths as possible to maximize code coverage. Fig. 1 highlights the high-level overview of coverage-guided fuzzing, consisting of stages like seed

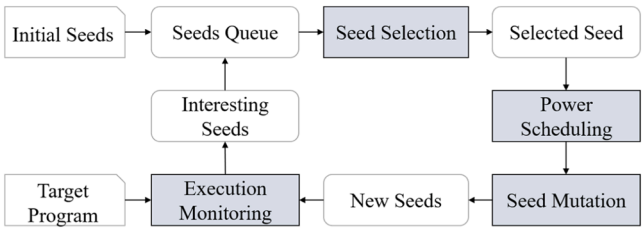


Fig. 1. High-level overview of coverage-guided fuzzing.

[1] We use PTFuzz instead of PTfuzz in the original paper to unify the form.

selection, power scheduling, seed mutation, and execution monitoring.

More specifically, (1) the actual fuzzing process starts with a set of user-provided initial seeds and maintains a seed queue. (2) *Seed selection* employs predetermined selection criteria to seek promising seed from seed queue. (3) *Power scheduling* allocates proper energy for selected seed. (4) *Seed mutation* applies mutation strategy on selected seed to yields new test inputs. (5) *Execution monitoring* runs target program on the yielded new test inputs to collect coverage information and monitor crashes. (6) It only retains and sends interesting test inputs to seed queue on the basis of coverage information (e.g., the ones that exercise new path) for the next round of fuzzing. (7) It returns to step 2 and continually repeats above process (Manès et al., 2019; Zhu et al., 2022). In this paper, we primarily concentrate on the step 2 and 3, i.e., seed selection and power scheduling.

A coverage-guided fuzzer usually uses the coverage information to decide which seeds should to fuzz next (i.e., seed selection) and how to specify the number of mutations (i.e., power scheduling). For example, AFL prioritizes seeds that are likely smaller or faster in the seed queue by collecting execution information to assess yielded new seeds. For selected seed, AFL primarily uses execution time and code coverage to calculate the potential of that seed, which can be used to determines how many new test inputs need to be generated, namely energy, in seed mutation stage. This energy is controlled with a power scheduling (Böhme et al., 2016).

2.2. Particle swarm optimization (PSO)

Particle Swarm Optimization (PSO) is a state-of-the-art yet simple swarm intelligence technique, inspired by social behavior of bird flocking or fish schooling (Eberhart and Kennedy, 1995). The PSO is analogous to a Genetic Algorithm (GA) in that the system is initialized with a population of random solutions and seeks for optimal solution by updating generations. Unlike GA, however, PSO assigns a randomized velocity for each potential solution (i.e., particle) and flies through the problem space by tracking the current optimum particles iteratively.

Algorithm 1 shows the procedure of PSO. Each particle i first initializes relevant parameters like its velocity v_i, position x_i, and calculates its fitness (Line 1–4). During each iteration, each particle moves towards its (1) local best position $lbest_i$ which are associated with the best fitness it has achieved so far (Line 3), and (2) global best position *gbest* which are gained by all particles up till now (Line 5). Hence, the velocity v_i, position x_i, and fitness of moved particle need to be updated on the basis of $lbest_i$ and *gbest* (Line 7–9). In addition, PSO will renovate $lbest_i$ and *gbest* utilizing the recalculated fitness for the next iteration (Line 10–15).

As the PSO iterates, each particle is drawn toward a global optimum via the interaction of their individual and colonial searches with a large probability. Therefore, PSO can iteratively reach the global optimum without getting stuck in local optimum. Also, PSO is easy to implement and lightweight, so it has rapidly progressed in recent years and with many successful applications in solving real-world optimization problems. Inspired by its successful application, we frame seed selection in fuzzing as a particle swarm optimization problem, in which each particle corresponds to a seed, and the selection of optimal position in each round is consistent with the selection of seeds. We believe that lightweight PSO algorithm could guide the fuzzer to efficiently choose the optimal seed as the next seed to be mutated, thereby improving the efficiency of fuzzing.

Algorithm 1
Particle Swarm Optimization (PSO).

```
Require: particle swarm N, velocity v_i, position x_i, local best position lbest_i, global best position gbest
1    for each i∈N do
2    initialize (v_i, x_i)
3    lbest_i = max (fitness (x_i))
4    end for
5    gbest = max (lbest_i)
6    while true do
7    for i from 1 to N do
8    update (v_i, x_i)
9    update_fitness (x_i)
10   if fitness (x_i) > fitness (lbest_i) then
11   lbest_i = x_i
12   end if
13   if fitness (lbest_i) > fitness (gbest) then
14   gbest = lbest_i
15   end if
16   end for
17   end while
```

2.3. Motivation

The efficiency of fuzzer may vary significantly depending on the seed selection criteria. Intuitively, different selection criteria work quite differently on one or more target applications. They are likely to perform well in certain programs but poorly in others. To verify our hypothesis, we evaluate several frequently-used selection criteria on AFL, covering code coverage, number of crashes, mutation depths, whether discover new paths, seed size, and execution time, namely, afl-cover, afl-crash, afl-depth, afl-path, afl-size, and afl-time in Fig. 2. Fig. 2 shows the number of unique paths and unique crashes during the fuzzing process by using single selection criterion on pdftotext and podofotxtextract. Each experiment runs for 12 h. In the light of Fig. 2, we can observe the following deduction.

Distinct selection criterions' efficiency differs in one target application. Apparently, the unique paths and unique crashes found rely on the seed selection criterion adopted by fuzzer. For instance, afl-cover, afl-path and afl-time could trigger approximately 2 × more unique crashes on podofotxtextract over other criteria like afl-crash, afl-depth and afl-size.

Different evaluation metrics' performance varies for one selection criterion. Another crucial observation is that the unique paths may be extremely prominent on one selection criterion, while its unique crashes are mediocre, and vice versa. For example, afl-depth could bring better unique paths on pdftotext over other criteria, while its number of unique crashes is relatively low.

Each selection criterion's efficiency varies with target applications. A selection criterion could perform well on one application, yet fail on another one. The afl-size, for example, could bring forth decent unique paths or unique crashes on pdftotext, whereas only 15 unique crashes can be discovered on podofotxtextract.

As we can note, in general, there are great differences in efficiency among diverse seed selection criteria. If we empirically integrate these selection criteria on the basis of their contribution, we may perform seed selection more efficiently. In this paper, we manage to select optimal seed via integrated criteria. To this end, we conduct fuzzing using single criterion to determine how much contribution a certain selection criterion makes. In the next section, we provide details on our proposed technique.

3. SYNTONY

In this section, we explain our design decisions to realize SYNTONY. Fig. 3 depicts an overview of SYNTONY. We aim at achieving high-efficiency fuzzing by utilizing potential-aware multi-criteria seed selection and power scheduling. To this end, SYNTONY first obtains the native or auxiliary properties (i.e., criteria) of seeds provided by executing target program along with these seeds from seeds queue. These criteria can be integrated to measure the quality of seeds across the board, allowing us to effectively characterize seed potential. Then, SYNTONY employs synthetical seed potential as selection principle to achieve potential-aware PSO selection model, unlike existing approaches that select seeds in a single principle. By doing this, SYNTONY could select the most promising next seed for fuzzing and hence achieve a more efficient seed selection strategy. Thanks to its simple algorithm,

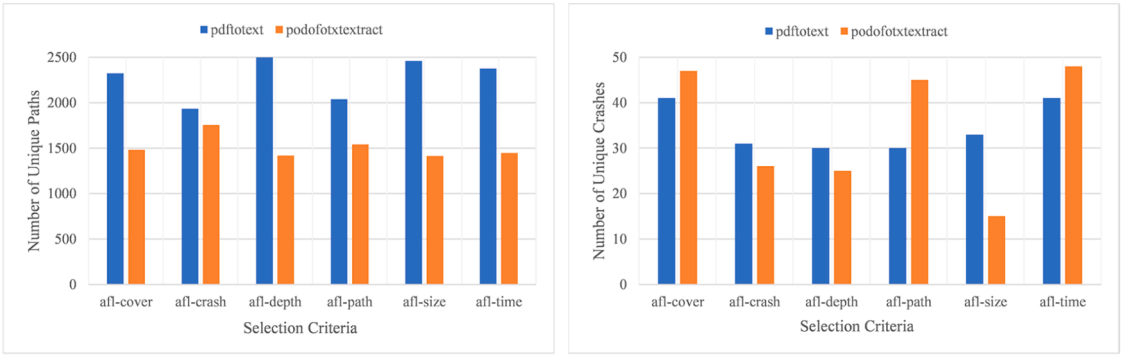


Fig. 2. Efficiency of various selection criteria on pdftotext and podofotxtextract in 12 h.

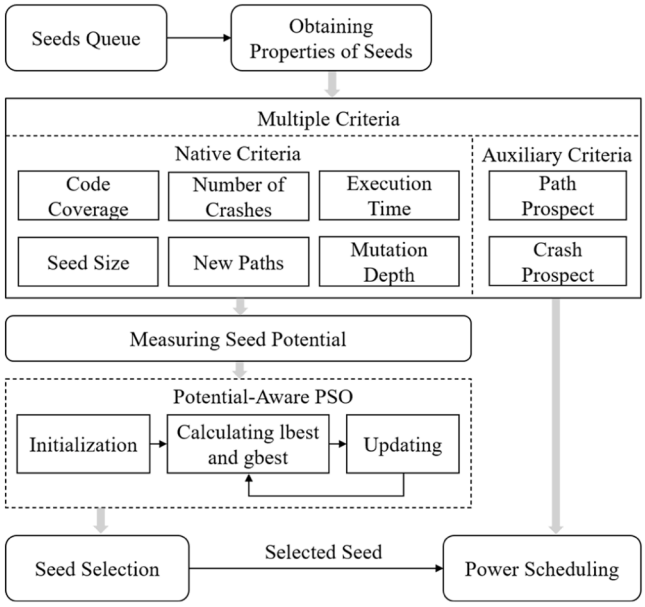


Fig. 3. An overview of SYNTONY.

SYNTONY can quickly perform fuzzing instead of relying on the heavyweight program analysis. To further improve the productivity of fuzzer, SYNTONY leverages diverse seed potential to facilitate power scheduling. More specifically, SYNTONY applies various properties that signify the prospect of seed to assign proper energy for selected seed. By prioritizing a seed with more potential as well as allocating more mutation energy to it, SYNTONY greatly increases the likelihood of discovering new paths or crashes. The rest of this section depicts the concrete details of our approaches.

3.1. Multiple criteria

We explain in detail eight native or auxiliary criteria of seeds to gain seed potential for fuzzing. Note that we employ the first six native criteria to achieve PSO-based multi-criteria seed selection strategy, and use the last two auxiliary criteria to improve power scheduling.

Execution time. Seeds with shorter execution time are preferred. Intuitively, faster seeds are more likely to cause far more fuzz runs during a campaign, and thus potentially more bugs.

Seed size. Smaller seeds tend to perform better than larger ones. The intuition is that more compact seeds probably increase the average speed of fuzzing and generate more interesting seed through mutation.

Code coverage. Seeds with higher code coverage may be preferable to one with lower coverage, like most coverage-guided fuzzers. Apparently, mutating a seed with nicer coverage will ultimately bring forth new coverage.

Whether the seed discovers new paths. Seeds covering new paths should have higher priority. The rationale behind new paths is similar to code coverage—a seed covering new paths is more likely to explore more uncovered new paths.

Number of crashes. Seeds triggering more crashes are usually treated with high priority as well. More tellingly, mutating a buggy seed tends to find more bugs and consequently increases the effectiveness of fuzzing.

Mutation depth. The mutation depth of seed refers to the mutation generations of seed. Seeds with deeper mutation generations are prioritized since fuzzing such seeds may trigger new paths or vulnerabilities.

Path prospect. Path prospect refers to the ratio of the descendants that discover a new path to the total number of descendants among the mutated descendants of a seed, which can be used as a pivotal factor to predict the probability of one seed discovering new paths.

Crash prospect. Likewise, crash prospect indicates the proportion of seed descendants that trigger a crash to the total number of descendants. It can be served as a major predictor for evaluating the likelihood of a seed finding a crash.

3.2. Seed potential

For each seed, we measure its potential p_i by a weighted sum method. Here we use six native criteria, i.e., execution time, seed size, code coverage, whether the seed discovers New Paths, number of crashes, and mutation depth, where the values of the first two criteria need to be converted to their opposites. Note that, each criterion should be further divided by the total value of all seeds for that criterion to explore optimum in probability space. For instance, we adopt the proportion of execution time to the cumulative time of all seeds to calculate p_i. Specifically, the seed potential p_i can be calculated as shown in the following equation.

$$p_i = \sum_{j=1}^{6} W_j \cdot \frac{C_j}{S_j} \tag{1}$$

Where C_j is the value of criterion j for seed i, and S_j is the cumulative value of all seeds in the seed queue for the criterion j. W_j is the weight of criterion j, which can be determined via existing objective weighting methods like EWM (i.e., Entropy Weight Method) (Klir and Folger, 1990) or CRITIC (i.e., Criteria Importance Through Intercriteria Correlation) (Diakoulaki et al., 1995).

EWM (Klir and Folger, 1990) is a widely-used information weighting method which determines objective weight for criteria via the differences between them. These differences are measured by information-theoretic entropy. CRITIC was originally proposed by Diakoulaki et al. (1995) to achieve relatively objective weights in MCDM (Multiple Criteria Decision Making). It determines objective weight through standard deviation of criteria and the correlation between criteria and other criteria. Compared with diverse subjective weighting

methods, these objective methods could avoid the interference of human factors on weight of criteria. In this paper, SYNTONY employs CRITIC to compute the weight of each criterion due to its better performance in the first step of experiment. Then, seed potential p_i that integrates six native criteria of seeds can be measured by Eq. (1).

3.3. Potential-aware PSO

In this section, we frame seed selection as a particle swarm optimization problem, in which each particle corresponds to a seed. Additionality, we employ seed potential computed by multiple selection criteria to achieve potential-aware PSO selection model that designates which seeds to choose in next round of fuzzing. Before elaborating on the potential-aware seed selection strategy, we first introduce three crucial options of PSO algorithm.

Particles. We assign a particle to per seed in the seed queue, and set its position x_i, velocity v_i, local best position $lbest_i$, and global best position *gbest* for each seed. During seed selection, the current position x_i of a particle (i.e., seed) in the probability space denotes the probability of selecting this seed for mutation, and the velocity v_i governs the direction in which the particle moves across the probability space. In each iteration, the position x_i of each particle will be updated on the basis of its v_i, $lbest_i$, *gbest*, and then attempt to move towards its optimal selection probability, which could yield more potential seeds. That particle with optimal position is the selected seed to be mutated. Specifically, the position x_i and velocity v_i of a particle at current iteration t are calculated as shown in the following equations.

$$v_i^t = wv_i^{t-1} + c_1 r_1 \left(lbest_i^{t-1} - x_i^{t-1}\right) + c_2 r_2 \left(gbest^{t-1} - x_i^{t-1}\right) \tag{2}$$

$$x_i^t = x_i^{t-1} + v_i^t \tag{3}$$

Where w is the inertia weight, which makes the particles maintain the inertia of motion and has a tendency to expand the search space. The acceleration constants c_1,c_2 are also called "cognitive coefficient" and "social coefficient", which are used to modulate the learning rate of the individual cognitive component and the group social component, respectively. r_1,r_2 are two uniformly distributed random numbers independently generated within [0, 1]. From Eq. (2), we can conclude that the motion direction of particle in the next iteration (i.e., velocity v_i) is contributed by three terms: inertial direction, individual optimal direction, and group optimal direction.

Local best position. Similar to original PSO, SYNTONY applies local best position $lbest_i$ to represent the optimal position that a particle i has ever explored during all iterations. More specifically, we always record the largest value of potential p_i for each particle (i.e., seed) in all rounds, and leverage this value to measure local optimal position $lbest_i$. Hence, $lbest_i$ is the position of the particle with the largest p_i in all iterations.

Global best position. Likewise, we specify the global best position *gbest* found by all particles during all iterations until now from a global perspective. The global best position *gbest* is assessed by seed potential p_i as well. In each iteration, SYNTONY regards the position with the maximal value of potential p_i among all particles as the global optimal position. Intuitively, *gbest* is the most potential position among $lbest_i$ so far.

3.4. Seed selection

SYNTONY aims to consecutively select the next seed with the greatest potential for mutation to optimize the efficiency of fuzzing. Algorithm 2 provides a general overview of potential-aware seed selection strategy. More specifically, SYNTONY utilizes aforementioned potential-aware PSO selection model to choose most promising seed which are more likely to generate interesting new seeds via mutation.

Like other coverage-guided fuzzers, SYNTONY maintains a seed queue that contains user-supplied initial seeds and new interesting seeds

Algorithm 2
Potential-aware selection.

Require: Seed queue: Q; The set of seed property: C;
The set of seed potential: P

```
1   while In the fuzzing loop do
2     for each i ∈ Q do
3       initialize_pso (x_i, v_i, lbest_i, gbest)
4       C_i = obtain_properties (i)
5       P_i = calculate_potential (C_i)
6       lbest_i = max (P_i)
7     end for
8     gbest = max (lbest_i)
9     for i from 1 to Q do
10      update_pso (x_i, v_i, lbest_i, gbest)
11    end for
12    next = select_optimum (Q, x_i)
13  end while
```

generated by mutation. The seeds then are chosen or mutated in a continuous fuzzing loop until a timeout is reached or the fuzzing is aborted (Line 1–13). For each seed i, SYNTONY initializes relevant parameters to set its position x_i, velocity v_i, local best position $lbest_i$, and global best position *gbest* (Line 3), and obtains its native properties C_i (Line 4) to further compute its potential P_i through summing multiple weighted criteria (Line 5). SYNTONY prefers to more promising seeds, i.e., the position of a seed x_1 is superior to that of another seed x_2, if and only if, the potential p_1 of the former is greater than the latter for a specific particle swarm. Consequently, SYNTONY chooses the maximal value of potential P_i that a particle has ever found as $lbest_i$ (Line 6), while the *gbest* is the optimum among $lbest_i$ (Line 8). Next, SYNTONY enters the loop of PSO responsible for updating the position x_i and velocity v_i of each seed using Eqs. (2) and (3) on the basis of $lbest_i$ and *gbest* (Line 10). After iteration, SYNTONY traverses the whole particle swarm, and selects particle with optimal position as the next seed to be mutated (Line 12). For selected seed, SYNTONY performs the mutation process of fuzzing according to defined mutation operators within the range of energy provided by power scheduling. If the generated new test inputs are considered to be interesting through executing target program, it will be added to the circular queue. At this point, SYNTONY re-traverses seed queue to iteratively choose the next seed.

3.5. Power scheduling

For each seed, the fuzzer determines its energy, which specifies the number of seeds generated by mutating that seed. This energy is controlled by a so-called power scheduling. Most coverage-guided fuzzers like AFL simply use the execution time, code coverage, and mutation depth to calculate the energy of seeds. On the contrary, SYNTONY affiliates two auxiliary criteria, i.e., path prospect and crash prospect, to computes the potential score for seeds.

In what follows, we define our potential-aware power scheduling utilizing the potential score of seeds as shown in Eq. (4). Given the seed i, SYNTONY optimizes its mutation energy based on the power scheduling of AFL as shown in Eq. (5).

$$score(i) = \beta \cdot path(i) + \delta \cdot crash(i) + \gamma \cdot \frac{cov(i) - cov(o)}{cov(o)} \cdot \frac{T}{time(i)} \tag{4}$$

Where $path(i)$ is path prospect while $crash(i)$ represents crash prospect, and β, δ, γ are constants. $cov(i)$ refers to the current code coverage, $cov(o)$ signifies the original code coverage when seed i has not been executed, $time(i)$ denotes the execution time of i, T is the total time of all seeds, and their combination becomes the expected coverage.

$$energy(i) = \sigma \cdot score(i) \cdot depth(i) \tag{5}$$

Where $score(i)$ is the potential score of seed i optimized on the basis of the original strategy, $depth(i)$ signifies the mutation depth of seed i, and the constant σ refers to the initial energy for each seed.

The combined potential score enables the fuzzer to fully exploit the prospect of seeds in both exploring paths and triggering crashes. As a result, SYNTONY assigns more energy for seeds with prominent potential to discover more new paths or crashes, thereby making our scheme more efficient.

4. Implementation

In this section, we provide a brief overview of the proof-of-concept implementation of our approach in the prototype of SYNTONY. We build SYNTONY on top of several mainstream open-source fuzzers (e.g., AFL, AFL++, and FairFuzz). SYNTONY retains the original mutation strategy of baseline fuzzer, but only changes seed selection and power scheduling in Fig. 1. Here we take AFL as an example to introduce the implementation of SYNTONY.

4.1. Seed selection module

We optimize the order in which AFL selects the seeds from the queue and how fuzzer determines "promising" seeds that are effectively exclusively chosen from the queue. More importantly, we leverage PSO algorithm to prioritize seeds and appoint seed potential p_i to guide seed selection. SYNTONY always chooses the seed at the optimal position as the next seed to mutate.

PSO initialization. SYNTONY first initializes several pivotal parameters for PSO, including particle position x_i, velocity v_i, local best position $lbest_i$, and global best position *gbest*. Specifically, SYNTONY sets the initial position x_i for each particle with a random value, and normalizes the sum of the positions for all particles to 1. The initial velocity v_i of each particle is set to 0.1. Also, SYNTONY initializes the local best position $lbest_i$, and global best position *gbest* to 0.5.

PSO updating. Each particle in the swarm needs to update the values of its position x_i and velocity v_i at each iteration to obtain the optimal position. The update method follows Eqs. (2) and (3). To this end, we first compute the local best position $lbest_i$ and global best position *gbest* with seed potential p_i. SYNTONY measures seed potential p_i that integrates six native properties of seeds via Eq. (1) to achieve multi-criteria seed selection strategy. To avoid subjective bias, we carry out fuzzing using single native property, and employ a previous approach, i.e., CRITIC (Diakoulaki et al., 1995), to obtain proper weights according to the experimental results. Then the local best position $lbest_i$ can be calculated by choosing the maximum value of the seed potential p_i for particle i in all iterations, while the global best position *gbest* is the maximum value of $lbest_i$. Also, we set the acceleration constants c_1,c_2 in Eq. (2) to 2 to balance the convergence speed and search effect. The inertia weight w is adjusted using a liner variation strategy (Shi and Eberhart, 1998), i.e., the inertia weight decreases linearly with the iterative generations. This strategy makes PSO focus on global optimization in the early stage, and focus on local optimization in the later stage.

Seed selection. After updating iteration, SYNTONY could approach to an optimal probability distribution for seeds. We choose seed with optimal probability (i.e., position x_i) as the next seed to be mutated, thereby improving the efficiency of fuzzing. More specifically, SYNTONY first traverses the position with the optimal probability, obtains its location index, and then searches the corresponding seed in the seed queue. The selected optimal seed will continue to perform subsequent power scheduling and seed mutation, and then repeat the above selection process.

4.2. Power scheduling module

We optimize the computation of the amount of fuzz, i.e., *energy*(i), designed for a seed i. AFL calculates *energy*(i) depending on execution time, code coverage, and mutation depth. Intuitively, AFL allocates more energy for a seed if it executes more quickly, covers more, and reaches deeper. SYNTONY maintains these native properties and further supplements two auxiliary properties, i.e., path prospect and crash prospect, to characterize the potential score of seeds. To this end, SYNTONY measures four metrics: (1) the total number of mutated descendants of a specific seed, (2) the number of its descendants that discover a new path, (3) the number of its descendants that trigger crash, (4) the original code coverage when it has not been executed, by instrumenting target programs. As a consequence, the number of fuzz, i.e., *energy*(i), can be computed by utilizing its potential score, following Eqs. (4) and (5). Essentially, SYNTONY prefers to assign more energy to seeds with higher potential scores, so that the fuzzer could yield interesting seeds with a greater probability.

5. Evaluation

To evaluate SYNTONY, this section aims at answering the following research questions.

- **RQ 1**: Are multi-criteria selection strategies better than single selection?
- **RQ 2**: How does SYNTONY compare against baseline fuzzer?
- **RQ 3**: Is SYNTONY compatible with other advanced fuzzers?
- **RQ 4**: How does SYNTONY compare to other state-of-the-art fuzzers?
- **RQ 5**: Which weighting methods performed best?
- **RQ 6**: How is the performance overhead of SYNTONY?

Among these 6 research questions above, RQ 1 tries to verify the effect of multi-criteria selection against single selection and RQ 5 emphasizes the measurement process of seed potential. Moreover, RQ 2 and RQ 4 manage to evaluate the effectiveness of SYNTONY that utilizes potential-aware seed selection strategy and optimized power scheduling strategy. RQ 3 further assesses the compatibility of SYNTONY. Also, RQ 6 focuses on evaluating the performance overhead of potential-aware seed selection strategy (i.e., PSO—Only) and SYNTONY.

5.1. Experimental setup

Experimental environment. All experiments are performed on a machine running 64-bit Ubuntu 16.04 LTS equipped with an Intel i7-6700 processor and 8GB RAM. Each experiment runs for 24 h.

Dataset. We evaluate SYNTONY on 14 popular real-world programs as shown in Table 1, each of which comes from different projects receiving a wide range of inputs, such as widely-used GNU binutils, pdf parsers, and diverse image parsers. We choose these 14 programs primarily for the following reasons. First, these employed datasets have various functionalities and diverse inputs. Second, most datasets adopted in our experiments are broadly used to evaluate other state-of-the-art fuzzers as well (Jain et al., 2018; Gan et al., 2020; Lyu et al., 2019; Zhang et al., 2018). Thus, our datasets are representative and

Table 1
Real-world programs used in evaluation.

Program	Project	Version	Input type
size	binutils	2.23	binary
strings	binutils	2.23	binary
jhead	jhead	2.97	jpg
sam2p	sam2p	0.49.4	bmp
gif2png	gif2png	3.0.0	gif
pngfix	libpng	1.6.34	png
w3m	w3m	0.5.3	text
png2swf	swftools	0.9.2	png
pdftohtml	xpdf	4.00	pdf
pdfimages	xpdf	4.00	pdf
pdftotext	xpdf	4.00	pdf
podofopdfinfo	podofo	0.9.6	pdf
podofopages	podofo	0.9.6	pdf
podofotxtextract	podofo	0.9.6	pdf

comprehensive. Additionality, we utilize a standard benchmark, i.e., LAVA-M dataset (Dolan-Gavitt et al., 2016), to examine the performance of fuzzers.

Baseline fuzzers. To compare the effectiveness of proposed SYNTONY, we choose several state-of-the-art fuzzers, covering AFL (Zalewski, 2019), AFL++ (Zalewski, 2020), MOPT (Lyu et al., 2019), FairFuzz (Lemieux and Sen, 2017), PTFuzz (Zhang et al., 2018), and MOAFL (Jiang et al., 2021), as our baselines. They are chosen based on the following considerations. We utilize AFL (Zalewski, 2019) because it is a well-known fuzzer and a good baseline for our evaluation. AFL++ (Zalewski, 2020) is an advanced fuzzer that integrates most of the improvements proposed recently. MOPT (Lyu et al., 2019) is a recently proposed state-of-the-art fuzzer with excellent performance. Also, we select FairFuzz (Lemieux and Sen, 2017) and PTFuzz (Zhang et al., 2018), two advanced fuzzers which are orthogonal to SYNTONY. Particularly, we apply SYNTONY to these two fuzzers and implement the prototypes SYNTONY, SYNTONY-FairFuzz, and SYNTONY-PTFuzz, to further highlight the effectiveness and compatibility of our proposed scheme. Furthermore, we use MOAFL (Jiang et al., 2021), our previous work, as it is related to the idea of SYNTONY. The pivotal difference between MOAFL and SYNTONY is the criteria and model for seed selection. The former employs native criteria and MOPSO to model seed selection, while the latter utilizes fine-grained auxiliary criteria and classic PSO to model seed selection and power scheduling.

5.2. Selection strategies (RQ 1)

To show how effectively our multi-criteria selection strategy can assist fuzzer in discovering new crashes or new code paths, we measured individually the number of unique crashes and paths during the fuzzing process by using multi-criteria selection strategy and six single-criterion selection strategy. All these selection strategies are deployed on AFL and run for 24 h with the same initial seeds and same settings as Section 5.1. Here we chose LAVA-M dataset as a fuzzing target because it has become a de-facto standard for evaluating fuzzers. It consists of four buggy programs (i.e., who, uniq, base64, and md5sum) that were manually injected with bugs. The results are summarized in Table 2. As expected, we see that for both unique paths and crashes, multi-criteria strategy significantly outperforms other single-criterion strategies.

For all the programs tested, multi-criteria strategy explores more unique paths than single-criterion strategies. For example, multi-criteria fuzzer discovers 869 unique paths in total, which increases by 35.78 % and 31.27 % over AFL-Cover and AFL-Time, respectively. Furthermore, multi-criteria strategy is quite effective in finding unique crashes as well. We observe that multi-criteria fuzzer, for instance, triggers significantly more unique crashes for base64 and md5sum — 69, 17, respectively — compared to 2, 6 unique crashes found by AFL-Depth. Especially in uniq, only multi-criteria fuzzer reports the unique crashes yet other fuzzers fail. This result is encouraging: multi-criteria does play an important role in finding unique paths or crashes.

Additionally, we can see that the difference between any two selection criteria is considerable. More importantly, there is no single winner that beats everyone else. For instance, AFL-Size shows vast potential on md5sum, whereas it performs moderately on base64. Even for AFL-Crash, although it discovers the largest amount of unique crashes in total, its unique paths are not significant. Hence, each selection criterion can make its own and unique contribution. This observation further motivates us to study the combination of diverse selection criteria.

5.3. Effectiveness evaluation (RQ 2)

To evaluate the effectiveness of our potential-aware scheme, we compared SYNTONY with the baseline fuzzer AFL++ using 11 representative real-world programs in Table 2. We measure the discovered edges, unique paths, and unique crashes during the fuzzing process as shown in Table 3. To avoid randomness, we conduct ten trials for each item, with each trial runs for 24 h. Table 3 shows the average value of edges, unique paths, and unique crashes found by SYNTONY and AFL++. As we can see, in all cases SYNTONY provides a significant advantage over the baseline.

Edges and Paths. SYNTONY outperforms AFL++ in terms of edges and unique paths; SYNTONY explored 28.69 % more edges and 25.02 % more unique paths than AFL++ in all programs. For example, SYNTONY finds nearly 27.38 % more edges over AFL++ on size. Similarly, for programs strings, gif2png, and size, SYNTONY achieves more code coverage — 110.04 %, 80.68 %, and 67.48 % more paths respectively, whereas AFL++ explores relatively few paths. This shows that some of the seeds selected by AFL++ fail to explore more and deeper code paths. These gains demonstrate the power of SYNTONY to choose potential seeds and allocate decent energy for them to explore the new paths or edges.

To further illustrate the efficiency of AFL++ vs SYNTONY, we also tracked the growth trend of the maximum and minimum number of edges explored within 24 h in ten repeated experiments, as shown in Fig. 4. The y-axis of each plot shows the cumulative number of edges and x-axis of plot denotes the time. From Fig. 4, we see that SYNTONY basically achieves the upper bound in edge coverage on repeated trials of all programs, especially on size, strings, podofopdfinfo, podofopages, and podofotxtextract. More importantly, the edges of SYNTONY still has an obvious growth trend at 24 h on pdftotext and strings, which implies that they will reach a higher upper bound if we continue to fuzz. Furthermore, SYNTONY could cover more edges in a faster pace than AFL++ in most programs such as size, strings, and podofopdfinfo.

Unique crashes. SYNTONY is quite effective on all programs in discovering unique crashes, except sam2p and gif2png in which SYNTONY and AFL++ have almost identical number of unique crashes. Interestingly, for programs jhead and pngfix, only SYNTONY reports the unique crashes yet AFL++ fails. More importantly, SYNTONY outperforms AFL++ at most 3200 % in these 11 programs. Overall, SYNTONY triggers 304 unique crashes in total, which increases by 132.06 % over AFL++. These results further support our design hypothesis that multi-criteria PSO selection model could efficiently choose promising seeds, which would benefit crash discovery and path exploration.

Furthermore, we plot time vs the maximum and minimum number of unique crashes triggered by AFL++ and SYNTONY in ten repeated experiments, as shown in Fig. 5. Note that, we only show the applications where two fuzzers found a crash. The y-axis of each plot shows the cumulative number of unique crashes and x-axis of plot denotes the time. As shown in Fig. 5, we could find SYNTONY trigger more unique crashes

Table 2
Number of paths and crashes found by various selection strategies on LAVA-M.

Programs	AFL-Cover		AFL-Crash		AFL-Depth		AFL-Path		AFL-Size		AFL-Time		Multi-Criteria	
	Paths	Crashes	Paths	Crashes	Paths	Crashes	Paths	Crashes	Paths	Crashes	Paths	Crashes	Paths	Crashes
who	176	2	187	3	182	2	172	2	195	2	183	2	207	3
uniq	88	0	87	0	93	0	92	0	90	0	93	0	95	2
base64	136	5	172	66	150	2	125	3	141	0	140	0	176	69
md5sum	240	6	245	6	250	6	375	8	388	8	246	6	391	17
Total	640	13	691	75	675	10	764	13	814	10	662	8	869	91

Table 3
The average of edges, paths, and crashes for AFL++ vs. SYNTONY on 11 programs.

Programs	AFL++			SYNTONY					
	Edges	Paths	Crashes	Edges	Increase	Paths	Increase	Crashes	Increase
size	3327	1021	19	4238	+27.38 %	1710	+67.48 %	38	+100 %
strings	2234	807	1	3669	+64.23 %	1695	+110.04 %	33	+3200 %
jhead	494	335	0	495	+0.20 %	357	+6.57 %	10	+10
sam2p	1212	895	0	1236	+1.98 %	988	+10.39 %	0	+0 %
gif2png	7748	409	0	10,100	+30.36 %	739	+80.68 %	0	+0 %
pngfix	1076	416	0	1137	+5.67 %	501	+20.43 %	12	+12
					=++		=++		=++
pdfimages	7774	3188	56	9396	+20.86 %	3716	+16.56 %	87	+55.36 %
pdftotext	9811	3960	53	11,457	+16.78 %	4452	+12.42 %	92	+73.58 %
podofopdfinfo	7973	1263	0	10,666	+33.78 %	1394	+10.37 %	2	+2
					=++		=++		=++
podofopages	8467	1330	1	10,780	+27.32 %	1606	+20.75 %	2	+100 %
podofotxtextract	8529	1445	1	12,299	+44.20 %	1682	+16.40 %	28	+2700 %
Total	58,645	15,069	131	75,473	+28.69 %	18,840	+25.02 %	304	+132.06 %

in a faster pace than baseline in several programs like size and strings. Also, SYNTONY takes fewer than 15 h to discover more unique crashes than AFL++ does in 24 h on most programs. After 15 h, for example, SYNTONY finds approximately 2 × more unique crashes than AFL++ on size. In summary, SYNTONY significantly outperforms AFL++ by achieving 0.20–64.23 % more edges coverage, 6.57–110.04 % more unique paths, and 55.36–3200 % more unique crashes across different programs.

Vulnerability discovery. To further evaluate the capability of SYNTONY in discovering vulnerabilities, we deduplicated and analyzed the crashes triggered in Fig. 5 to confirm whether a real vulnerability was found. More specifically, we recompiled the aforementioned programs with AddressSanitizer and rechecked them with the seeds that triggered the crash. Then we manually analyzed the location of vulnerability, and compare it with the corresponding vulnerability report found on the CVE website to further confirm the CVE. The vulnerabilities discovered by AFL++ and SYNTONY are shown in Table 4.

As we can see, SYNTONY outperforms AFL++ in discovering real vulnerabilities. After deduplication, SYNTONY found CVEs in 7 programs, while AFL++ found CVEs in 5 of them. For instance, the vulnerability CVE-2020-6624 in jhead is a heap buffer overflow vulnerability, which occurs in the function *process_DQT*() of the file *jpgqguess.c*, at line 109. More importantly, for programs strings and pdftotext, only SYNTONY reports the vulnerabilities yet AFL++ fails. Overall, SYNTONY finds 13 CVEs in total on these applications, which increases by 116.67 % over AFL++.

5.4. Compatibility evaluation (RQ 3)

To evaluate the compatibility of SYNTONY, we applied our scheme to multiple coverage-guided fuzzers like AFL (Zalewski, 2019), FairFuzz (Lemieux and Sen, 2017) and PTFuzz (Zhang et al., 2018), and implemented the prototypes SYNTONY-AFL, SYNTONY-FairFuzz, and SYNTONY-PTFuzz based on their counterparts. Here we chose 9 frequently-used open-source programs, namely, jhead, sam2p, gif2png, pngfix, w3m, png2swf, pdftohtml, podofopdfinfo, and podofopages, to evaluate SYNTONY-AFL, SYNTONY-FairFuzz, SYNTONY-PTFuzz, and their counterparts. The result is summarized in Table 5, Fig. 6, and Fig. 7. Overall, SYNTONY-based fuzzers significantly prevail over their counterparts by finding more crashes or paths on most cases.

For all the programs, SYNTONY-based fuzzers significantly outperform their counterparts in finding more unique paths, as shown in Table 5 and Fig. 6. For example, SYNTONY-FairFuzz explores around 1.3 × more unique paths than FairFuzz on gif2png. Furthermore, compared to their counterparts, the trends of path increments explored by SYNTONY-based fuzzers are roughly the same. For instance, the number of unique paths found by SYNTONY-FairFuzz and SYNTONY-PTFuzz increase approximately 1.23 × and 1.14 × over their counterparts on pngfix. In total, SYNTONY-AFL, SYNTONY-FairFuzz, and SYNTONY-PTFuzz can effectively improve the code coverage over AFL, FairFuzz, and PTFuzz by 12.35 %, 10.08 %, and 22.10 %.

As shown in Fig. 7, SYNTONY-based fuzzers can efficiently discover unique crashes for these binaries as well. Unfortunately, none of these fuzzers found crashes on sam2p, thus we only show the programs where these fuzzers found a crash. Interestingly, SYNTONY-FairFuzz surprisingly discovers 75 unique crashes on w3m, which increases by 12.5 × over FairFuzz. SYNTONY-PTFuzz triggers up to 5 unique crashes on w3m, where PTFuzz discovers no crash at all. Overall, SYNTONY-AFL, SYNTONY-FairFuzz, and SYNTONY-PTFuzz find on average 89.31 %, 200 %, and 64.8 6% more crashes than AFL, FairFuzz, and PTFuzz, respectively. Thus, our scheme is compatible with a variety of coverage-guided fuzzers and enables these fuzzers to explore more code paths and trigger more unique crashes.

5.5. Further comparison (RQ 4)

To further evaluate the effectiveness of our scheme, we also compared SYNTONY with other state-of-the-art fuzzers (i.e., MOPT (Lyu et al., 2019), FairFuzz (Lemieux and Sen, 2017), PTFuzz (Zhang et al., 2018), and MOAFL (Jiang et al., 2021)) using 9 real-world programs in Table 1. MOPT (Lyu et al., 2019), a recently introduced state-of-the-art fuzzer that utilizes Particle Swarm Optimization (i.e., PSO) to guide AFL's mutation strategy. FairFuzz (Lemieux and Sen, 2017) also builds on top of AFL, which mutates preferentially rare branches with mutation mask. PTFuzz (Zhang et al., 2018) leverages hardware mechanism Intel PT to facilitate information collection of fuzzing. MOAFL (Jiang et al., 2021) is our previous work that employs novel Multi-Objective Particle Swarm Optimization (i.e., MOPSO) to choose seeds. Since MOPT, FairFuzz, PTFuzz, and MOAFL are all deployed on AFL, we use SYNTONY-AFL for comparison in this section.

Table 6 shows the number of unique paths and crashes found by SYNTONY, FairFuzz, PTFuzz, MOPT, and MOAFL. Also, Fig. 8 presents the growth trend of unique paths discovered by these fuzzers. Compared to other advanced fuzzers, SYNTONY shows a strong and stable growth trend in discovering unique paths on most programs like w3m and podofopdfinfo. Especially, when fuzzing gif2png and pdftohtml, SYNTONY discovers about 2.34 × and 4.04 × more program paths than FairFuzz and PTFuzz in 24 h, respectively. For sam2p and podofopdfinfo, SYNTONY could explore unique paths faster, and explore much more. Overall, SYNTONY finds around 24.38 %, 32.27 %, 10.09 %, and 6.53 % more unique paths than FairFuzz, PTFuzz, MOPT, and MOAFL.

Additionally, SYNTONY significantly outperforms FairFuzz, PTFuzz, MOPT, and MOAFL in finding unique crashes in most cases. For instance, SYNTONY finds 67.65 % more crashes than FairFuzz on png2swf, and discovers 21.28 % more crashes than MOPT. Especially in jhead, only

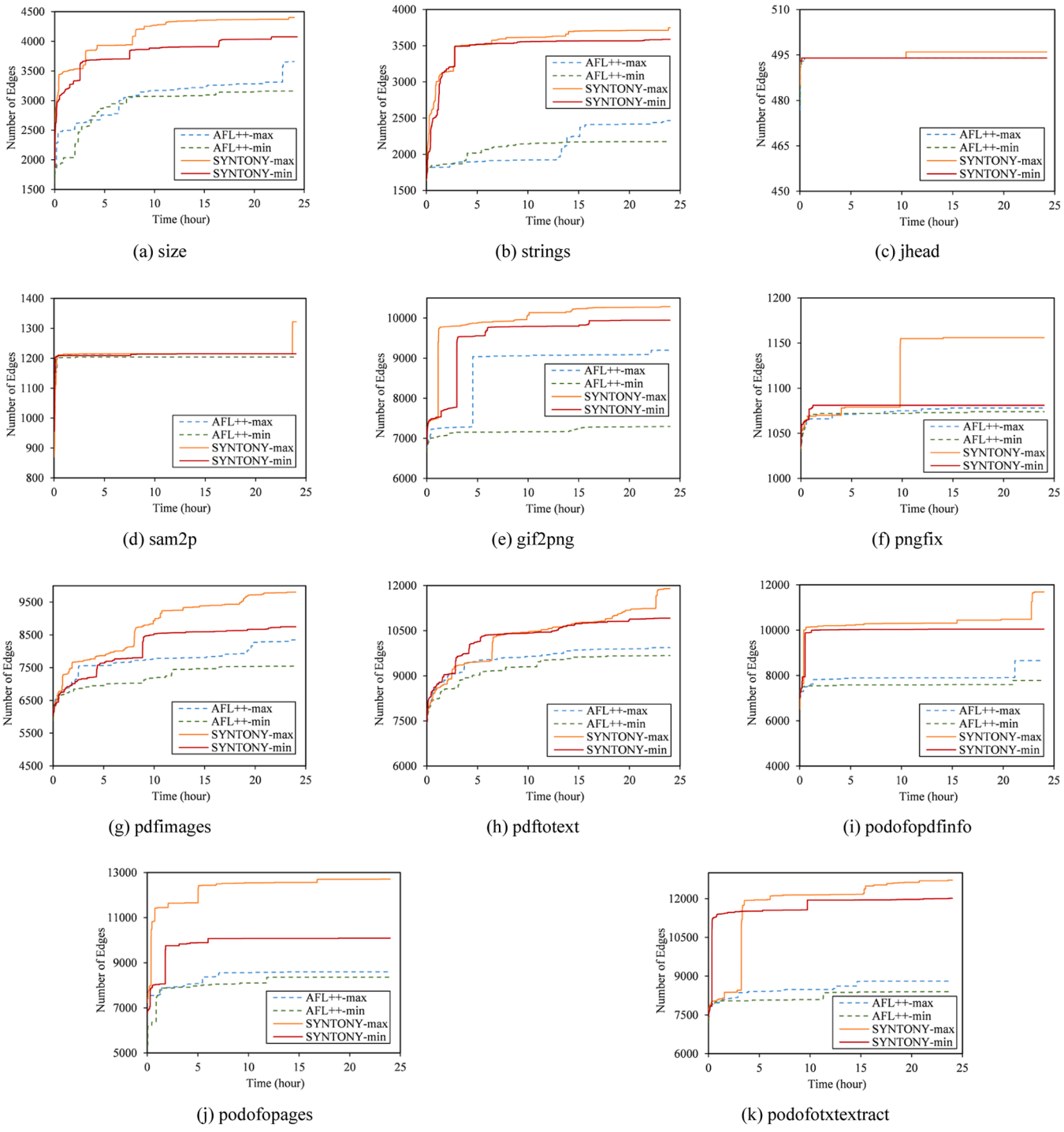


Fig. 4. Edges over time for AFL++ vs. SYNTONY in 24 h.

SYNTONY reports the unique crashes yet other fuzzers fail. However, SYNTONY, like the other three fuzzers, cannot find unique crashes for sam2p. In total, the number of unique crashes found by SYNTONY exceeds FairFuzz, PTFuzz, MOPT, and MOAFL nearly 5.51 × , 6.70 × , 2.82 × , and 2.82 × on these programs.

5.6. *Weighting effect (RQ 5)*

To show the effect of employed weighting method (i.e., CRITIC), we compared SYNTONY with a fuzzer that uses weights provided by EWM. Here we still deploy the proposed method on AFL. Table 7 summarizes the number of unique paths and unique crashes found by SYNTONY-CRITIC and SYNTONY-EWM on 9 widely-used real-world programs. For each target, we performed an experiment with the same setup as the previous experiments (again, 24 h). We see that for both unique paths and unique crashes, SYNTONY-CRITIC and SYNTONY-EWM significantly outperform baseline fuzzer.

For all the programs, SYNTONY-CRITIC and SYNTONY-EWM achieve more unique paths than AFL, on average by 12.21 % and 10.03 %. Also, SYNTONY-CRITIC still surpasses SYNTONY-EWM in exploring unique paths on each program except pdfimages. Overall, SYNTONY-CRITIC has a mild advantage over SYNTONY-EWM. Fig. 9 visualizes

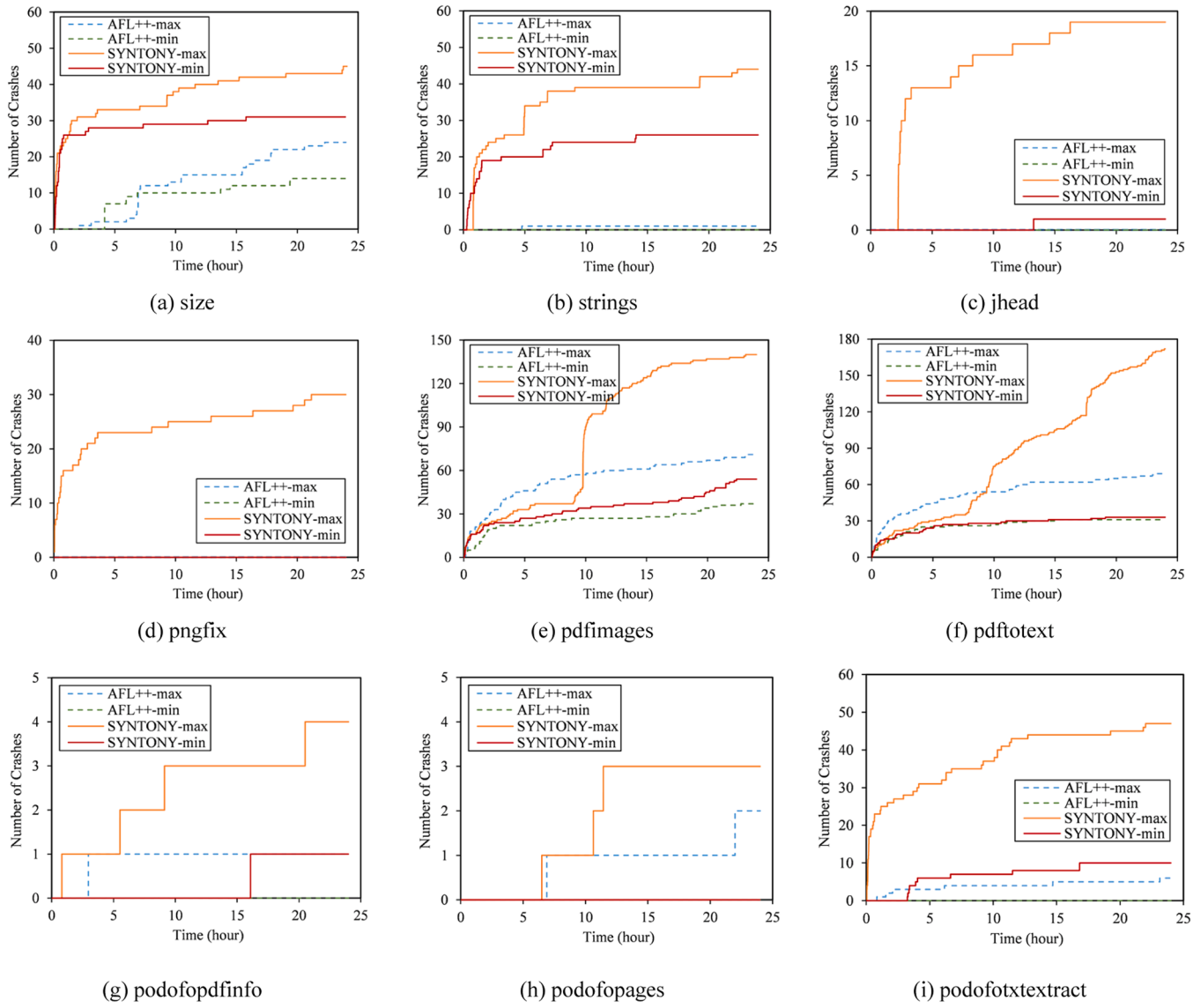


Fig. 5. Crashes over time for AFL++ vs. SYNTONY in 24 h.

Table 4
Vulnerabilities discovered by AFL++ vs. SYNTONY.

Programs	Types	AFL++	SYNTONY
size	stack buffer overflow	CVE-2014-8503; CVE-2014-8504	CVE-2014-8503; CVE-2014-8504
	out-of-bounds read		CVE-2014-8484
strings	stack buffer overflow		CVE-2014-8503; CVE-2014-8504
	out-of-bounds read		CVE-2014-8484
jhead	heap buffer overflow	CVE-2020-6624	CVE-2020-6624
pdfimages	stack overflow	CVE-2022-38334	CVE-2022-38334
pdftotext	stack overflow		CVE-2022-38334
podofopdfinfo	memory allocation violation	CVE-2018-5783	CVE-2018-5783; CVE-2019-10723
	excessive recursion		CVE-2018-11254
podofopages	stack overflow	CVE-2021-30470	CVE-2021-30470

the results of the unique paths found over time. We see that on most applications, SYNTONY-CRITIC and SYNTONY-EWM achieve the upper bound in unique paths, especially for sam2p, pngfix, png2swf, and pdfimages, generally showing the effectiveness of path exploration.

More tellingly, SYNTONY-CRITIC and SYNTONY-EWM are effective at discovering new unique crashes for binaries as well. Especially in gif2png and pngfix, only SYNTONY-CRITIC and SYNTONY-EWM report the unique crashes while AFL finds no crash at all. Overall, SYNTONY-CRITIC and SYNTONY-EWM discover 326 and 231 unique crashes in total, which significantly increases by 102.48 % and 43.48 % over AFL, respectively. In addition, SYNTONY-CRITIC observably outperforms SYNTONY-EWM in terms of the unique crashes. Among these 9 programs, SYNTONY-CRITIC on average triggers 41.13% more unique crashes than SYNTONY-EWM. Therefore, the weighting method employed (i.e., CRITIC) is much more efficient in finding unique crashes and exploring code paths than EWM.

5.7. Overhead (RQ 6)

Obviously, the PSO model may mildly bring additional overhead for fuzzers, thereby reducing execution speed. To show the overhead of SYNTONY, we used 6 programs to compare the average number of

Table 5
The unique paths and crashes found by various fuzzers on 9 programs.

Programs	AFL		SYNTONY-AFL		FairFuzz		SYNTONY-FairFuzz		PTFuzz		SYNTONY-PTFuzz	
	Paths	Crashes	Paths	Crashes	Paths	Crashes	Paths	Crashes	Paths	Crashes	Paths	Crashes
jhead	344	0	351	7	345	0	354	0	347	0	366	18
sam2p	828	0	1000	0	864	0	871	0	830	0	877	0
gif2png	1087	0	1188	4	508	0	669	0	1182	0	1475	0
pngfix	494	0	636	26	496	0	611	0	464	0	529	0
w3m	2058	92	2108	144	1801	6	1878	75	1537	0	1747	5
png2swf	849	36	1002	57	711	34	966	42	768	37	782	38
pdftohtml	1070	1	1379	3	1318	1	1393	7	341	0	347	0
podofopdfinfo	1405	0	1538	3	1276	0	1406	1	1239	0	2162	0
podofopages	1566	2	1697	4	1444	4	1498	10	1532	0	1776	0
Total	9701	131	10,899	248	8763	45	9646	135	8240	37	10,061	61

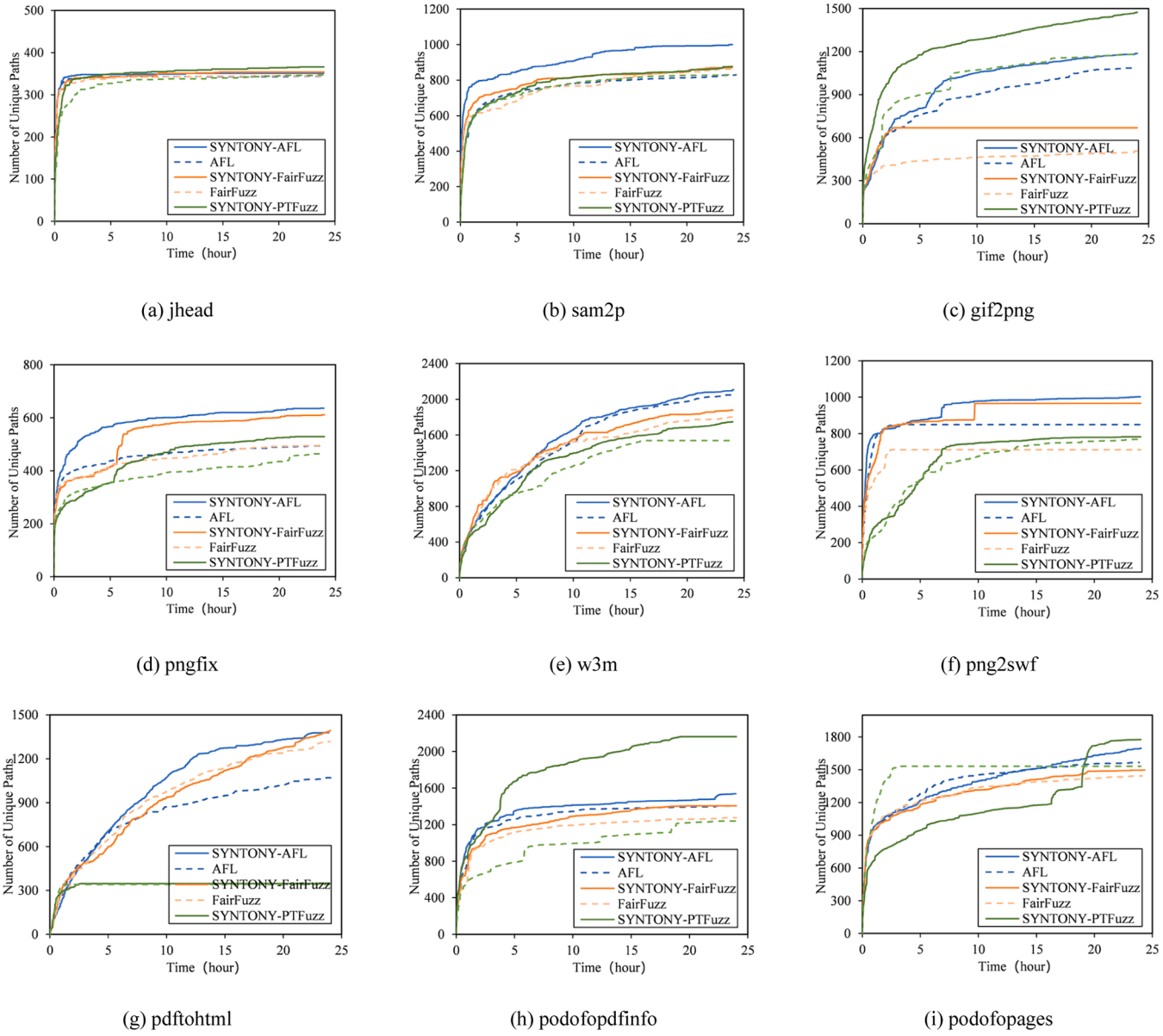


Fig. 6. Unique paths over time for various fuzzers in 24 h.

executions for SYNTONY with original AFL. Additionally, we employed PSO-only SYNTONY fuzzer without potential-aware power scheduling to further evaluate the execution efficiency of our proposed seed selection approach. Table 8 reports the average number of executions during the fuzzing process by using AFL, SYNTONY, and PSO-only SYNTONY, with each experiment runs for one hour. Among these programs, the executions of each programs are various. The fuzzers, for example, execute slowly in several large binaries like pdfimages, whereas others do not.

Although SYNTONY fuzzers take partial computing power to

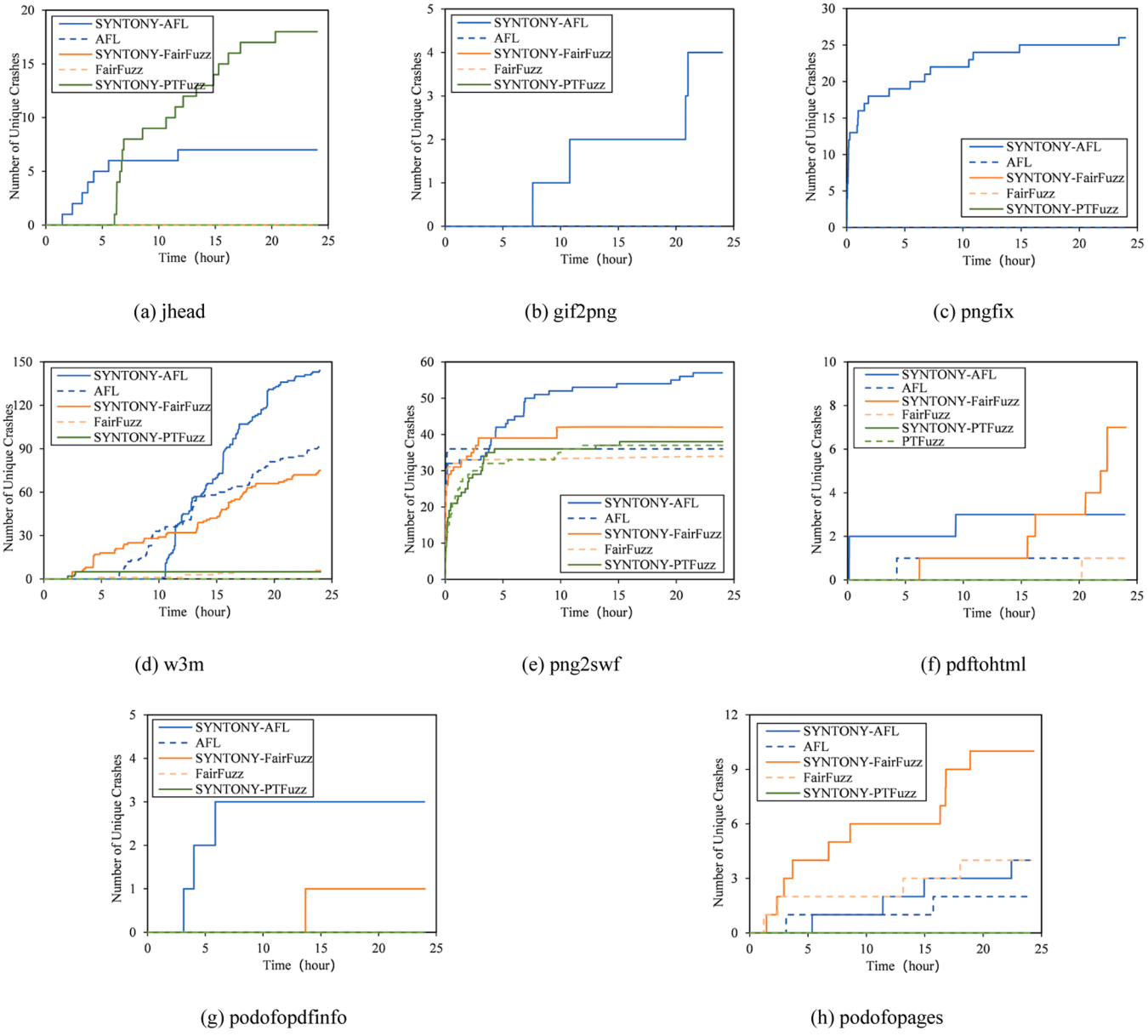


Fig. 7. Unique crashes over time for various fuzzers in 24 h.

Table 6
Number of unique paths and crashes found by various fuzzers on 9 programs.

Programs	SYNTONY		FairFuzz		PTFuzz		MOPT		MOAFL	
	Paths	Crashes	Paths	Crashes	Paths	Crashes	Paths	Crashes	Paths	Crashes
jhead	351	7	345	0	347	0	346	0	347	0
sam2p	1000	0	864	0	830	0	913	0	938	0
gif2png	1188	4	508	0	1182	0	1169	4	1153	0
pngfix	636	26	496	0	464	0	592	1	643	0
w3m	2108	144	1801	6	1537	0	1926	29	2001	48
png2swf	1002	57	711	34	768	37	922	47	945	37
pdftohtml	1379	3	1318	1	341	0	1218	0	1128	1
podofopdfinfo	1538	3	1276	0	1239	0	1325	3	1454	0
podofopages	1697	4	1444	4	1532	0	1489	4	1622	2
total	10,899	248	8763	45	8240	37	9900	88	10,231	88

improve the seed selection, the average executions of SYNTONY and PSO-only SYNTONY are still comparable with AFL on several programs, such as sam2p and gif2png. Overall, SYNTONY and PSO-only SYNTONY decreases 6.32 % and 5.55 % execution efficiency than AFL, respectively. Considering the increased efficiency in discovering crashes and paths, such slight speed reduction introduced by SYNTONY is moderate and acceptable. Moreover, PSO-only SYNTONY can execute the tests slightly faster than SYNTONY in many cases, on average by

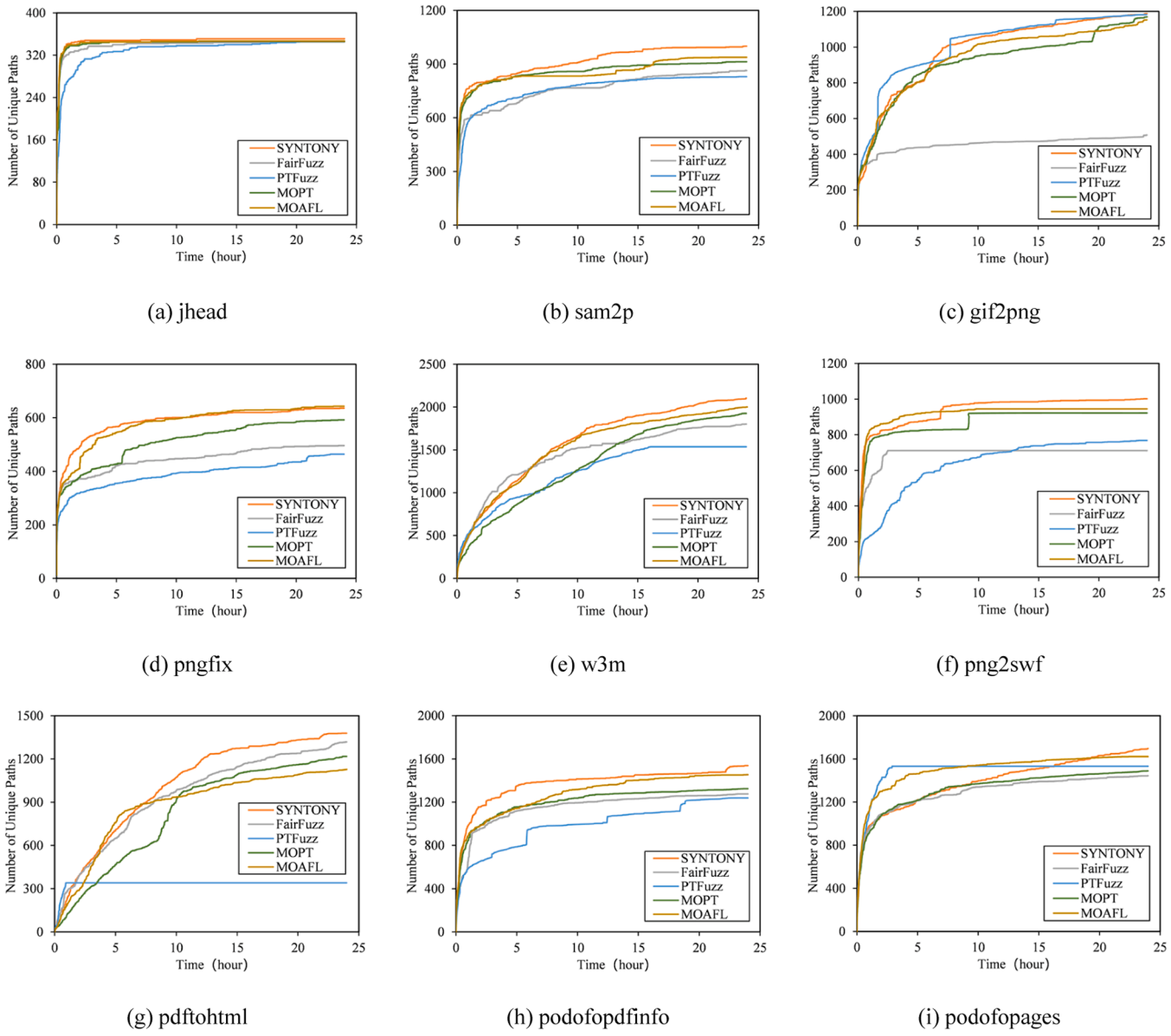


Fig. 8. Number of unique paths over time for various fuzzers in 24 h.

Table 7
The effect for CRITIC vs. EWM on 9 programs.

Programs	AFL		SYNTONY- CRITIC				SYNTONY-EWM			
	Paths	Crashes	Paths	Increase	Crashes	Increase	Paths	Increase	Crashes	Increase
sam2p	828	0	1000	+20.77 %	0	+0 %	980	+18.36 %	0	+0 %
gif2png	1087	0	1188	+9.29 %	4	+4	1101	+1.29 %	2	+2
pngfix	494	0	636	+28.74 %	26	+26	621	+25.71 %	21	+21
w3m	2058	92	2108	+2.43 %	144	+56.52 %	2087	+1.41 %	119	+29.35 %
png2swf	849	36	1002	+18.02 %	57	+58.33 %	992	+16.48 %	37	+2.78 %
pdftohtml	1070	1	1379	+28.88 %	3	+200 %	1189	+11.12 %	1	+0 %
pdfimages	2416	30	2663	+10.22 %	85	+183.33 %	2885	+19.41 %	45	+5 %
podofopdfinfo	1405	0	1538	+9.47 %	3	+3	1446	+2.92 %	0	+0 %
podofopages	1566	2	1697	+8.37 %	4	+100 %	1653	+5.56 %	6	+200 %
Total	11,773	161	13,211	+12.21 %	326	+102.48 %	12,954	+10.03 %	231	+43.48 %

0.82 %. This result demonstrates that the additional overhead of SYNTONY primarily lies in the PSO selection model, whereas the power scheduling hardly introduces any cost, empirically showing that the performance of SYNTONY can be further improved if we optimize the PSO model. In summary, while SYNTONY introduces additional overhead due to our PSO selection model, such overhead is still moderate and acceptable.

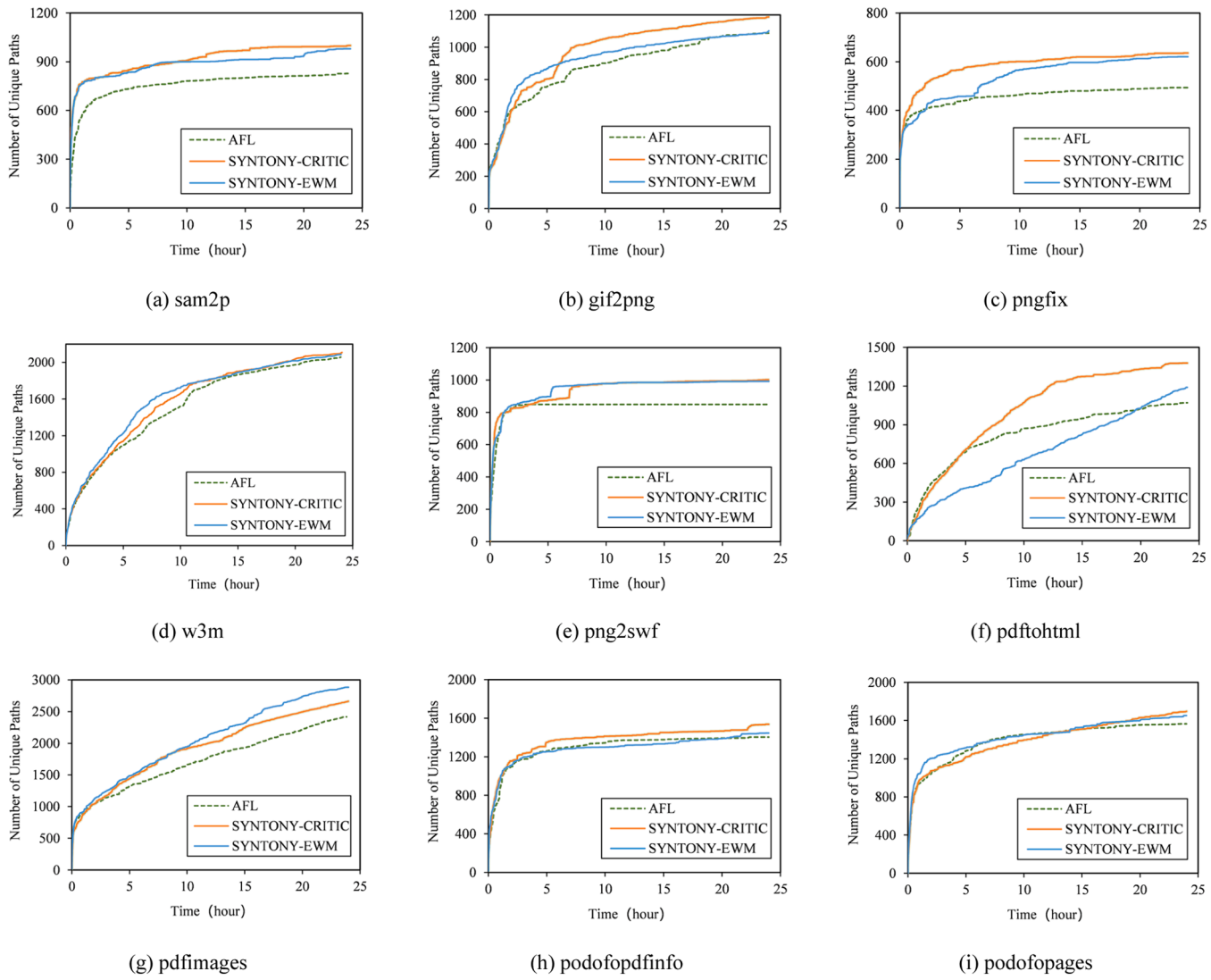


Fig. 9. Paths over time for AFL, SYNTONY, and SYNTONY-EWM in 24 h.

Table 8
Average number of executions for various fuzzers on 6 programs (executions per second).

Programs	AFL	SYNTONY	PSO—Only
size	271.55	265.71	267.75
jhead	186.41	175.12	177.50
sam2p	174.70	170.82	174.48
gif2png	33.52	30.13	33.16
pngfix	1073.86	988.14	992.92
pdfimages	27.01	25.45	23.22
average	294.51	275.90	278.17

6. Discussion & future work

We discuss the compatibility of SYNTONY, the limitations of SYNTONY, and future direction of research.

Complementing with other fuzzers. Our proposed SYNTONY are orthogonal to a wide range of existing fuzzers that attempt to enhance AFL, such as FairFuzz (Lemieux and Sen, 2017), PTFuzz (Zhang et al., 2018), GREYONE (Gan et al., 2020), and NAUTILUS (Aschermann et al., 2019a). Therefore, SYNTONY can complement the others by effectively prioritizing hopeful seeds and allocating more energy for them. We believe our seed selection strategy and power scheduling strategy could be used in conjunction with fuzzers targeting the mutation or other issue to build a more effective fuzzer. On the contrary, other state-of-the-art strategies can be further applied to SYNTONY as a supplement. For instance, hardware mechanism in PTFuzz, byte-level mutation in GREYONE, and grammar-based seed generation in NAUTILUS should be well integrated with the proposed scheme to show better results.

Limitations and future works. Although effective, SYNTONY still has a lot to be improved. Currently, SYNTONY employs original PSO algorithm with additional weighting factor to achieve multi-criteria seed selection strategy. Nevertheless, this method may be limited in accuracy if sample data is incomplete. We thus plan to overcome this limitation by leveraging advanced multi-objective algorithm to adaptively explore seed potential for specific target program. More importantly, SYNTONY currently uses only several simple seed properties, such as execution time, code coverage, and mutation depth, all of which are essential for measuring seed potential. It would be worth exploring how to apply other fine-grained selection criteria like the ratio of effective byte to extend SYNTONY's testing capability. Additionally, there are several interesting research directions that might increase fuzzing efficiency even more. As an example, we plan to integrate existing advanced mutation strategies to further enhance our general-purposed fuzzer.

7. Related work

In this section, we briefly summarize the existing fuzzing mechanisms and test case prioritization techniques, i.e., TCPs.

Coverage-guided fuzzing. Recent advantages in coverage-guided fuzzing has shown very promising results in security testing (Serebryany, 2016; Swiecki, 2019; Zalewski, 2019). It usually utilizes coverage as a feedback mechanism to seek which inputs are interesting and which do not trigger new behavior or transition (Aschermann et al., 2019b). More importantly, AFL (Zalewski, 2019) becomes one of the most well-recognized fuzzers thanks to its ease-of-use. Inspired by the large number of bugs found by AFL, research recently focused heavily on optimizing AFL using state-of-the-art techniques (Stephens et al., 2016; Böhme et al., 2016; Rawat et al., 2017; Lemieux and Sen, 2017). FairFuzz (Lemieux and Sen, 2017) uses program analysis and heuristics to discover rare branches and mutate inputs hitting rare branches via mask. Steelix (Li et al., 2017) infers magic bytes utilizing static analysis and extra instrumentation. In addition, a recent study, T-Fuzz (Peng et al., 2018), transforms target programs to bypass complex sanity checks and reproduces true bugs in the original program.

Taint analysis could track critical bytes that affect program execution in the input to determine which part of the input should be modified (Chen and Chen, 2018; Wang et al., 2010; Haller et al., 2013; Neugschwandtner et al., 2015). Vuzzer (Rawat et al., 2017) develops application-aware mutation strategy by performing static analysis and dynamic taint analysis. TIFF (Jain et al., 2018) infers the type of input bytes utilizing data-structure identification and taint analysis to guide mutation. More recently, GREYONE (Gan et al., 2020) adopts fuzzing-driven taint inference to infer taint of variables.

Several fuzzers make use of symbolic execution or concolic execution to increase code coverage and find bugs (Cadar et al., 2008; Cha et al., 2012; Godefroid et al., 2012; Cha et al., 2015). Driller (Stephens et al., 2016) combines fuzzing with selective concolic execution in a complementary manner. QSYM (Yun et al., 2018) is a more recent work that designs a concolic executor tailored for hybrid fuzzing to achieve better performance. Another recent work DigFuzz (Zhao et al., 2019) predicts the difficulty for a fuzzer to explore a path and assigns difficulty paths for concolic execution via Monte Carlo algorithm.

However, most of these taint-based and symbolic-based approaches struggle scale to large complicated programs due to heavyweight program analysis or path explosion. SYNTONY does not rely on any heavy program analysis techniques like taint analysis or symbolic execution, and thus can freely scale to large binaries. More importantly, a majority of these coverage-guided fuzzers are orthogonal to our solution. As a result, SYNTONY can be complemented by such approaches due to its well-deserved compatibility.

Seed selection strategies. A crucial optimization for coverage-guided fuzzers is to choose the first-rank seed input wisely. Several papers and security practitioners utilize heuristics to select seed inputs for fuzzing (Householder and Foote, 2012; Woo et al., 2013; Rebert et al., 2014). AFL (Zalewski, 2019) prefers smaller and faster seeds to fuzz more seed inputs in a specific time. AFLFast (Böhme et al., 2016) uses a Markov chain model to identify seeds exercising less-frequent paths. VUzzer (Rawat et al., 2017) leverage control-flow and data-flow features to prioritize the seeds whose path is hard-to-reach, and deprioritize seeds exercising error-handling paths. CollAFL (Gan et al., 2018) employs three different policies to prioritize seeds that exercise untouched neighbor branches or descendants, and Angora (Chen and Chen, 2018) selects the inputs whose paths contain conditional statements with unexplored branches.

On a different spectrum, directed fuzzing also preferentially selects seeds that could reach predetermined targets as soon as possible (Wang et al., 2016; Petsios et al., 2017; Chen et al., 2019b). AFLGo (Böhme et al., 2017) prioritizes seed inputs that gravitate towards the target locations via simulated annealing-based power scheduling, and Hawkeye (Chen et al., 2018) is prone to select seeds that hold the trace closer to the target sites using dynamic metrics. Furthermore, machine learning techniques can be used to identify the vulnerable sites of target programs (Wang et al., 2019a; Li et al., 2019b; Zong et al., 2020). NeuFuzz (Wang et al., 2019a) and V-Fuzz (Li et al., 2019b) give priority to seeds that are more likely to cover vulnerable paths under the guidance of neural network-based vulnerability prediction model. In contrast, FuzzGuard (Zong et al., 2020) deprioritizes the unreachable seeds predicted by deep learning.

While these efforts, existing approaches utilize a single-objective for seed prioritization, which may cause bias and starve certain promising seeds. To mitigate such a limitation, Cerebro (Li et al., 2019a) utilizes an online multi-objective optimization (MOO) model together with a non-dominated sorting algorithm to balance various metrics such as code complexity and coverage. Also, the complexity of uncovered code near the trace of input provided by static analyzer is applied to improve power scheduling. However, this online MOO model may sacrifice its performance, since calculating the Pareto frontier and revealing the convergence usually require more than 100 iterations. On the contrary, SYNTONY employ a lightweight Particle Swarm Optimization (PSO) for multi-criteria seed selection, so that it can efficiently select the most potential seed for detecting bugs in real-world binaries.

Fuzzing boosting. Recently, a large quantity of boosting techniques has been proposed to improve the efficiency of fuzzing from diverse aspects (Lyu et al., 2019; Aschermann et al., 2019b; You et al., 2019; Wang et al., 2019b; Chen et al., 2019c). Among them, MOPT (Lyu et al., 2019) searches the optimal distribution for mutation operators utilizing Particle Swarm Optimization. Unlike MOPT, SYNTONY leverage Particle Swarm Optimization to achieve multi-criteria seed selection. Also, the performance of fuzzers itself are improved in multiple ways (Zhang et al., 2018; Xu et al., 2017; Chen et al., 2019a; Nagy and Hicks, 2019; Song et al., 2019). Furthermore, it is worth mentioning that deep learning or reinforcement learning techniques have been extensively applied in improving the diverse stages of fuzzing (Godefroid et al., 2017; Rajpal et al., 2017; Karamcheti et al., 2018; Böttinger et al., 2018; Drozd and Wagner, 2018; She et al., 2019). More recently, several works such as Grimoire (Blazytko et al., 2019), NAUTILUS (Aschermann et al., 2019a), and Superion (Wang et al., 2019c) combine the ideas of generation-based fuzzing with coverage feedback to discover hard-to-trigger bugs that happen after lexical and syntactical checks. In contrary, ANTIFUZZ (Güler et al., 2019) and Fuzzification (Jung et al., 2019) protect the released binaries against fuzzing from a fire-new perspective.

Instead of focusing on mutation or generation stages, SYNTONY attempts to improve seed selection stage and power scheduling process for fuzzing to efficiently choose promising seeds that can discover more new paths or bugs. Taking the advantage of its compatibility, it can be easily integrated with a wide range of the aforementioned fuzzers.

TCP techniques. There are several previous works on reordering test cases in software testing, which involves test case prioritization techniques (TCPs) (Chen et al., 2017; Chen et al., 2016; Wang et al., 2017b; Spieker et al., 2017). TCPs are one of the predominant regression testing techniques, which prioritizes test cases to maximize a certain objective function, typically exploring faults earlier (Rothermel et al., 1999; Mukherjee and Patnaik, 2021). A majority of existing TCPs leverage code coverage information (Rothermel et al., 1999; Hao et al., 2016), historical information (Huang et al., 2012), requirement information (Hettiarachchi et al., 2014), cost or time information (Srikanth et al., 2009; You et al., 2011), and similarity between test cases and source code (Saha et al., 2015) to prioritize test cases. Also, QTEP (Wang et al., 2017b) adopts a typical statistic defect prediction model as well as a typical static bug finder to detect fault-prone code, and then uses weighted coverage information to prioritize test cases.

Instead of attempting to focus on single objective function, an alternative set of approaches utilizes multi-objective techniques to prioritizing test cases (Epitropakis et al., 2015; Panichella et al., 2015; Mao et al., 2016; Marchetto et al., 2016). To solve multi-objective TCP

problems, Panichella et al. (2015) propose a novel multi-objective genetic algorithm DIV-GA which integrates orthogonal design and orthogonal evolution. Alessandro Marchetto et al. (2016) determine test case orderings utilizing the coverage of source code, application requirements, and the execution cost of test cases to identify critical and fault-prone portions of software artifacts. Inspired by the above-mentioned approaches, we apply multi-criteria prioritization technique to seed selection stage of fuzzing.

8. Conclusion

This paper presented SYNTONY, a potential-aware fuzzing scheme that manages to optimize seed selection and power scheduling utilizing seed potential. To this end, SYNTONY first measures the potential of seeds via multiple integrated criteria to guide seed selection and power scheduling. SYNTONY then employs lightweight Particle Swarm Optimization (i.e., PSO) to achieve efficient potential-aware seed selection strategy without any heavyweight program analysis, which makes SYNTONY scalable enough to fuzz complex real-world binaries. Additionality, we adopt a variety of seed prospects to facilitate power scheduling. By doing this, SYNTONY can take full advantages of the potentiality of seeds and further efficiently prioritize promising seeds as well as allocate proper energy to them. We implemented the proposed scheme in multiple state-of-the-art fuzzers like AFL, AFL++, FairFuzz, and PTFuzz. Our evaluation results demonstrated that SYNTONY-based fuzzers significantly outperformed their counterparts in discovering unique crashes and paths on several real-world programs.

CRediT authorship contribution statement

Xiajing Wang: Conceptualization, Methodology, Software, Validation, Investigation, Resources, Data curation, Writing – original draft, Writing – review & editing, Funding acquisition. **Rui Ma:** Conceptualization, Methodology, Resources, Writing – original draft, Supervision, Project administration, Funding acquisition. **Wei Huo:** Conceptualization, Resources, Writing – review & editing, Supervision, Funding acquisition. **Zheng Zhang:** Software, Validation, Investigation, Writing – review & editing. **Jinyuan He:** Software, Validation, Visualization. **Chaonan Zhang:** Formal analysis, Visualization. **Donghai Tian:** Validation, Investigation.

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.

Data availability

Data will be made available on request.

Acknowledgments

We would like to thank all practitioners who participated in the focus group discussions, as well as We thank the anonymous reviewers for their constructive comments to improve the paper. This work was supported by the Opening Foundation of Key Laboratory of Network Evaluation Technology, CAS under Grant KFKT2022-008 and the Science Foundation for Young Scientists of OUC under Grant Q22F0027.

References

Aschermann, C., Frassetto, T., Holz, T., Jauernig, P., Sadeghi, A.-R., Teuchert, D., 2019a. NAUTILUS: fishing for deep bugs with grammars. In: Proceedings of 2019 Network and Distributed System Security Symposium, pp. 1–15.

Aschermann, C., Schumilo, S., Blazytko, T., Gawlik, R., Holz, T., 2019b. REDQUEEN: fuzzing with input-to-state correspondence. In: Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS). The Internet Society, pp. 1–15.

Blazytko, T., Aschermann, C., Schlögel, M., Abbasi, A., Schumilo, S., Wörner, S., et al., 2019. Grimoire: synthesizing structure while fuzzing. In: Proceedings of the 28th USENIX security symposium, pp. 1985–2002.

Böhme, M., Pham, V., Nguyen, M., Roychoudhury, A., 2017. Directed greybox fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS), ACM, pp. 2329–2344.

Böhme, M., Pham, V., Roychoudhury, A., 2016. Coverage-based greybox fuzzing as Markov chain. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS). ACM, pp. 1032–1043.

K. Böttinger, P. Godefroid, and R. Singh, Deep reinforcement fuzzing, pp. 116–122, 2018. arXiv:1801.04589. https://doi.org/10.48550/arXiv.1801.04589.

Cadar, C., Dunbar, D., Engler, D., 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX conference on Operating systems design and implementation, pp. 209–224.

Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D., 2012. Unleashing mayhem on binary code. In: Proceedings of IEEE Symposium on Security and Privacy (SP), pp. 380–394.

Cha, S.K., Woo, M., Brumley, D., 2015. Program-adaptive mutational fuzzing. In: Proceedings of the 2015 IEEE Symposium on Security and Privacy, pp. 725–741.

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 (CCS). ACM, pp. 2095–2108.

Chen, J., Bai, Y., Hao, D., Xiong, Y., Zhang, H., Xie, B., 2017. Learning to prioritize test programs for compiler testing. In: International Conference on Software Engineering (ICSE), pp. 700–711.

Chen, J., Bai, Y., Hao, D., Xiong, Y., Zhang, H., Zhang, L., Xie, B., 2016. Test case prioritization for compilers: a text-vector based approach. In: International Conference on Software Testing, Verification and Validation (ICST). IEEE, pp. 266–277.

Chen, P., Chen, H., 2018. Angora: efficient fuzzing by principled search. In: Proceedings of the 2018 IEEE Symposium on Security and Privacy (S&P). IEEE, pp. 711–725.

Chen, P., Liu, J., Chen, H., 2019c. Matryoshka: fuzzing deeply nested branches. In: Proceedings of the ACM Conference on Computer and Communications Security, pp. 499–513.

Chen, Y., Jiang, Y., Ma, F., Liang, J., Wang, M., Zhou, C., Jiao, X., Su, Z., 2019a. Enfuzz: ensemble fuzzing with seed synchronization among diverse fuzzers. In: Proceedings of the 28th USENIX Security Symposium, pp. 1967–1983.

Y. Chen, P. Li, J. Xu, S. Guo, and L. Lu, SAVIOR: towards bug-driven hybrid testing, pp.1–17, 2019. arXiv: 1906.07327.

Diakoulaki, D., Mavrotas, G., Papayannakis, L., 1995. Determining objective weights in multiple criteria problems: the critic method. Comput. Oper. Res. 763–770.

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: Proceedings of the 2016 IEEE Symposium on Security and Privacy (S&P). IEEE, pp. 110–121.

W. Drozd and M.D. Wagner, FuzzerGym: a competitive framework for fuzzing and learning, pp. 1–15, 2018. arXiv:1807.07490. https://doi.org/10.48550/arXiv.1807.07490.

Eberhart, R., Kennedy, J., 1995. A new optimizer using particle swarm theory. In: Sixth International Symposium on Micro Machine & Human Science (MHS), pp. 39–43.

Epitropakis, M.G., Yoo, S., Harman, M., Burke, E.K., 2015. Empirical evaluation of pareto efficient multi-objective regression test case prioritisation. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA), ACM, pp. 234–245.

Gan, S., Zhang, C., Chen, P., Zhao, B., 2020. GREYONE: data flow sensitive fuzzing. In: Proceedings of 29th USENIX Security Symposium, USENIX, pp. 1–18.

Gan, S., Zhang, C., Qin, X., Tu, X., Li, K., Pei, Z., Chen, Z., 2018. CollAFL: path sensitive fuzzing. In: Proceedings of the 2018 IEEE Symposium on Security and Privacy (S&P). IEEE, pp. 679–696.

Godefroid, P., Levin, M., Molnar, D., 2012. SAGE: whitebox Fuzzing for security testing. Commun. ACM 40–44.

Godefroid, P., Peleg, H., Singh, R., 2017. Learn&fuzz: machine learning for input fuzzing. In: Proceedings of 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE/ACM, pp. 50–59.

Güler, E., Aschermann, C., Abbasi, A., Holz, T., 2019. Antifuzz: impeding fuzzing audits of binary executables. In: Proceedings of the 28th USENIX Security Symposium, pp. 1–17.

Haller, I., Slowinska, A., Neugschwandtner, M., Bos, H., 2013. Dowsing for overflows: a guided fuzzer to find buffer boundary violations. In: Proceedings of the 22nd USENIX Security Symposium, pp. 49–64.

Hao, D., Zhang, L., Zang, L., Wang, Y., Wu, X., Xie, T., 2016. To be optimal or not in test-case prioritization. IEEE Trans. Softw. Eng. 42, 490–504.

Hettiarachchi, C., Do, H., Choi, B., 2014. Effective regression testing using requirements and risks. In: 8th IEEE International Conference on Software Security and Reliability (SERE), pp. 157–166.

Householder, A.D., Foote, J.M., 2012. Probability-based Parameter Selection for Black-box Fuzz Testing. Software Engineering Institute, pp. 1–30.

Huang, Y.C., Peng, K.L., Huang, C.Y., 2012. A history-based cost-cognizant test case prioritization technique in regression testing. J. Syst. Softw. 626–637.

Jain, V., Rawat, S., Giuffrida, C., Bos, H., 2018. TIFF: using input type inference to improve fuzzing. In: Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC), pp. 505–517.

Jiang, J., Ma, R., Wang, X., He, J., Tian, D., Li, J., 2021. MOAFL: potential seed selection with multi-objective particle swarm optimization. In: Proceedings of the 7th International Conference on Communication and Information Processing (ICCIP '21). Association for Computing Machinery, pp. 26–31.
Jung, J., Hu, H., Solodukhin, D., Pagan, D., Lee, K.H., Kim, T., 2019. Fuzzification: anti-fuzzing techniques. In: Proceedings of the 28th USENIX Security Symposium, pp. 1967–1983.
Karamcheti, S., Mann, G., Rosenberg, D., 2018. Adaptive grey-box fuzz-testing with Thompson sampling. In: Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security, ACM, pp. 37–47.
Klir, G.J., Folger, T.A., 1990. Fuzzy sets, uncertainty and information. J. Oper. Res. Soc. 26, 1–64.
C. Lemieux and K. Sen, FairFuzz: targeting rare branches to rapidly increase greybox fuzz testing coverage, pp. 1–11, 2017. arXiv: 1709.07101. Doi:10.48550/arXiv.1709.07101.
Li, Y., Chen, B., Chandramohan, M., Lin, S., Liu, Y., Tiu, A., 2017. Steelix: program-state based binary fuzzing. In: Joint Meeting on Foundations of Software Engineering, pp. 627–637.
Y. Li, S. Ji, C. Lv, Y. Chen, J. Chen, Q. Gu, and C. Wu, V-fuzz: vulnerability-oriented evolutionary fuzzing, pp.1–16, 2019. arXiv:1901.01142.
Li, Y., Xue, Y., Chen, H., Wu, X., Zhang, C., Xie, X., Wang, H., Liu, Y., 2019a. 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 (ESEC/FSE), pp. 533–544.
Lyu, C., Ji, S., Zhang, C., Li, Y., Lee, W., Song, Y., Beyah, R., 2019. MOPT: optimized mutation scheduling for fuzzers. In: Proceedings of 28th USENIX Security Symposium, USENIX, pp. 1–21.
Manès, V., Han, H., Han, C., Cha, S., Egele, M., Schwartz, E., Woo, M., 2019. The art, science, and engineering of fuzzing: a survey. IEEE Trans. Softw. Eng. (TSE'19) 2312–2331.
Mao, K., Harman, M., Jia, Y., 2016. Sapienz: multi-objective automated testing for android applications. In: Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA), ACM, pp. 94–105.
Marchetto, A., Islam, M.M., Asghar, W., Susi, A., Scanniello, G., 2016. A multi-objective technique to prioritize test cases. IEEE Trans. Softw. Eng. 918–940.
Mukherjee, R., Patnaik, K.S., 2021. A survey on different approaches for software test case prioritization. J. King Saud Univ. - Comput. Inf. Sci. 33, 1041–1054.
Nagy, S., Hicks, M., 2019. Full-speed fuzzing: reducing fuzzing overhead through coverage-guided tracing. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 787–802.
Neugschwandtner, M., Comparetti, P.M., Haller, I., Bos, H., 2015. The BORG: nanoprobing binaries for buffer overreads. In: Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, pp. 87–97.
Panichella, A., Oliveto, R., Penta, M.D., De Lucia, A., 2015. Improving multi-objective test case selection by injecting diversity in genetic algorithms. IEEE Trans. Softw. Eng. 5589, 1–28.
Peng, H., Shoshitaishvili, Y., Payer, M., 2018. T-Fuzz: fuzzing by program transformation. In: Proceedings of the 2018 IEEE Symposium on Security and Privacy (S&P). IEEE, pp. 1–14.
Petsios, T., Zhao, J., Keromytis, A.D., Jana, S., 2017. Slowfuzz: automated domain-independent detection of algorithmic complexity vulnerabilities. In: Conference on Computer and Communication Security, pp. 2155–2168.
M. Rajpal, W. Blum, and R. Singh, Not all bytes are equal: neural byte sieve for fuzzing, pp. 1–10, 2017. arXiv:1711.04596. Doi:10.48550/arXiv.1711.04596.
Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H., 2017. VUzzer: application-aware evolutionary fuzzing. In: Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), pp. 1–14.
Rebert, A., Cha, S.K., Avgerinos, T., Foote, J., Warren, D., Grieco, G., Brumley, D., 2014. Optimizing seed selection for fuzzing. In: Proceedings of the 23rd USENIX Security Symposium, pp. 861–875.
Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J., 1999. Test case prioritization: an empirical study. In: International Conference on Software Maintenance (ICSM), pp. 179–188.
Saha, R.K., Zhang, L., Khurshid, S., Perry, D.E., 2015. An information retrieval approach for regression test prioritization based on program changes. In: International Conference on Software Engineering (ICSE), pp. 268–279.
Serebryany, K., 2016. Continuous Fuzzing with Libfuzzer and Address Sanitizer. 2016 IEEE Cybersecurity Development, p. 157.
She, D., Pei, K., Epstein, D., Yang, J., Ray, B., Jana, S., 2019. NEUZZ: efficient fuzzing with neural program learning. In: Proceedings of the 2019 IEEE Symposium on Security and Privacy (S&P). IEEE, pp. 1–13.
Shi, Y., Eberhart, R.C., 1998. A modified particle swarm optimizer. In: IEEE International Conference on Evolutionary Computation Proceedings, pp. 69–73.
Song, C., Zhou, X., Yin, Q., He, X., Zhang, H., Lu, K., 2019. P-Fuzz: a parallel grey-box fuzzing framework. Appl. Sci. 9, 1–14.
Spieker, H., Gotlieb, A., Marijan, D., Mossige, M., 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, pp. 12–22.
Srikanth, H., Cohen, M.B., Qu, X., 2009. Reducing field failures in system configurable software: cost-based prioritization. In: In Proceedings of International Symposium on Software Reliability Engineering (ISSRE), pp. 61–70.
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: Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS). The Internet Society, pp. 1–16.
R. Swiecki, Honggfuzz. http://code.google.com/p/honggfuzz, 2019 (accessed 15 June 2019).
Wang, J., Chen, B., Wei, L., Liu, Y., 2017a. Skyfire: data-driven seed generation for fuzzing. In: Proceedings of the 2017 IEEE Symposium on Security and Privacy (S&P). IEEE, pp. 579–594.
Wang, J., Chen, B., Wei, L., Liu, Y., 2019c. Superion: grammar-aware greybox fuzzing. In: Proceedings of International Conference on Software Engineering, pp. 1–12.
Wang, J., Duan, Y., Song, W., Yin, H., Song, C., 2019b. Be sensitive and collaborative: analyzing impact of coverage metrics in greybox fuzzing. In: Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID), USENIX, pp. 1–15.
Wang, S., Nam, J., Tan, L., 2017b. QTEP: quality-aware test case prioritization. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 523–534.
Wang, T., Wei, T., Gu, G., Zou, W., 2010. TaintScope: a checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy, pp. 497–512.
Wang, W., Sun, H., Zeng, Q., 2016. Seededfuzz: selecting and generating seeds for directed fuzzing. In: 10th International Symposium on Theoretical Aspects of Software Engineering (TASE), pp. 49–56.
Wang, Y., Wu, Z., Wei, Q., Wang, Q., 2019a. NeuFuzz: efficient fuzzing with deep neural network. IEEE Access 36340–36352.
Woo, M., Cha, S.K., Gottlieb, S., Brumley, D., 2013. Scheduling black-box mutational fuzzing. In: Proceedings of the 2013 ACM Conference on Computer & Communications Security, pp. 511–522.
Xu, W., Kashyap, S., Kim, T., 2017. Designing new operating primitives to improve fuzzing performance. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS). ACM, pp. 2313–2328.
You, D., Chen, Z., Xu, B., Luo, B., Zhang, C., 2011. An empirical study on the effectiveness of time-aware test case prioritization techniques. In: Proceedings of the 2011 ACM Symposium on Applied Computing, pp. 1451–1456.
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: Proceedings of the 2019 IEEE Symposium on Security and Privacy (S&P). IEEE, pp. 1–18.
Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T., 2018. QSYM: a practical concolic execution engine tailored for hybrid fuzzing. In: Proceedings of the 27th USENIX Security Symposium, USENIX, pp. 745–761.
M. Zalewski, American Fuzzy Lop plus plus. https://github.com/AFLplusplus/AFLplusplus/, 2020 (accessed 15 June 2022).
Michal Zalewski, American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/, 2019 (accessed 15 June 2019).
Zhang, G., Zhou, X., Luo, Y., Wu, X., Min, E., 2018. PTfuzz: guided fuzzing with processor trace feedback. IEEE Access 37302–37313.
Zhao, L., Duan, Y., Yin, H., Xuan, J., 2019. Send hardest problems my way: probabilistic path prioritization for hybrid fuzzing. In: Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS). The Internet Society, pp. 1–15.
Zhu, X., Wen, S., Camtepe, S., Xiang, Y., 2022. Fuzzing: A Survey for Roadmap. ACM Computing Surveys, pp. 1–36.
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. In: Proceedings of the 2020 USENIX Security Symposium, pp. 1–15.

Xiajing Wang received the Ph.D. degree in software engineering from Beijing Institute of Technology, China, in 2021. Currently she is a lecturer at the Open University of China. Her research interests include information security and vulnerability discovery.

Rui Ma received the Ph.D. degree from Beijing Institute of Technology, China, in 2004. She is an associate professor with the School of Computer Science and Technology, Beijing Institute of Technology, China. Her current research interests include software security, Internet of things, neural network, and data mining.

Wei Huo received the Ph.D. degree from Chinese Academy of Sciences, China, in 2010. He is a professor with Institute of Information Engineering, Chinese Academy of Sciences, China. His-current research interests include vulnerability discovery, software security evaluation, and information system security.

Zheng Zhang received the B.E. degree in software engineering from the Beijing Institute of Technology, China, in 2022. He is currently pursuing the M.S. degree at Beijing Institute of Technology, China. His-research interests include software security and vulnerability analysis.

Jinyuan He received the B.E. degree in software engineering from the Beijing Institute of Technology, China, in 2019. She is currently pursuing the Ph.D. degree at Beijing Institute of Technology, China. Her research interests include software security and vulnerability analysis.

Chaonan Zhang received the Ph.D. degree in Control Science and Engineering from Beijing Institute of Technology, China, in 2019. She was a full-time postdoctoral fellow from 2019 to 2022 at the Department of Mechanical Engineering and Automation, Beihang

University, China. Currently she is a lecturer at the Open University of China. Her research interests are focused on the microrobots and artificial intelligence.

Donghai Tian received the Ph.D. degree from Beijing Institute of Technology, China, in 2012. He is a teacher with the School of Computer Science and Technology, Beijing Institute of Technology, China. His-current research interests include software security, malware analysis and detection, Android security, and cloud security.