MOPT: Optimize Mutation Scheduling for Fuzzers

Chenyang Lv[†], Shouling Ji[†], Chao Zhang[¶], Yuwei Li[†], Wei-Han Lee[§], Yu Song[†], and Raheem Beyah[‡]

[†]Zhejiang University
[¶] Tsinghua University
[§]IBM Research
[‡]Georgia Institute of Technology
E-mails: puppet@zju.edu.cn, sji@zju.edu.cn, chaoz@tsinghua.edu.cn, liyuwei@zju.edu.cn, wei-han.lee1@ibm.com, zjujakesong@gmail.com, rbeyah@ece.gatech.edu.

Abstract

Mutation-based fuzzing is one of the most popular vulnerability discovery solutions. Its performance of generating interesting test cases highly depends on the mutation scheduling strategies. However, existing fuzzers usually follow a specific (e.g., uniform) distribution to select mutation operators, which is inefficient in finding vulnerabilities on general programs. Thus, in this paper, we present a novel mutation scheduling scheme MOPT, which enables mutation-based fuzzers to discover vulnerabilities more efficiently. MOPT utilizes a customized Particle Swarm Optimization (PSO) algorithm to find the optimal selection probability distribution of operators with respect to fuzzing effectiveness, and provides a pacemaker fuzzing mode to accelerate the convergence speed of PSO. We applied MOPT to the state-of-the-art fuzzers AFL, AFLFast and Vuzzer, and implemented MOPT-AFL, -AFLFast and -VUzzer respectively, and then evaluated them on 13 real world open-source programs. The results showed that, MOPT-AFL could find 170% more security vulnerabilities and 350% more crashes than AFL. MOPT-AFLFast and MOPT-VUzzer also outperform their counterparts. Furthermore, the extensive evaluation also showed that MOPT provides a good rationality, compatibility and steadiness, while introducing negligible costs.

1 Introduction

Mutation-based fuzzing is one of the most prevalent vulnerability discovery solutions. In general, it takes seed test cases and selects them in certain order, then mutates them in various ways, and tests target programs with the newly generated test cases. Many new solutions have been proposed in the past years, including the ones that improve the seed generation solution [1, 2, 3, 4], the ones that improve the seed selection strategy [5, 6, 7, 8, 9], the ones that improve the testing speed and code coverage [10, 11, 12, 13], and the ones that integrate other techniques with fuzzing [14, 15, 16].

However, less attention has been paid to how to mutate test cases to generate new effective ones. A large number of well-recognized fuzzers, e.g., AFL [17] and its descendants, libFuzzer [18], honggfuzz [19], BFF [2] and VUzzer [7], usually predefine a set of *mutation operators* to characterize where to mutate (e.g., which bytes) and how to mutate (e.g., add, delete or replace bytes). During fuzzing, they use certain *mutation schedulers* to select operators from this predefined set, in order to mutate test cases and generate new ones for fuzzing. Rather than directly yielding a mutation operator, the mutation scheduler yields a probability distribution of predefined operators, and the fuzzer will select operators following this distribution. For example, AFL uniformly selects mutation operators.

There are limited solutions focusing on improving the mutation scheduler. Previous works [8, 9] utilize reinforcement learning to dynamically select mutation operators in each round. However, they do not show significant performance improvements in vulnerability discovery [8, 9]. Thus, a better mutation scheduler is demanded. We figure out that, most previous works cannot achieve the optimal performance because they fail to take the following issues into consideration.

Different operators' efficiency varies. Different mutation operators have different efficiency in finding crashes and paths (as shown in Fig. 3). Thus, fuzzers that select mutation operators with the uniform distribution are likely to spend unnecessary computing power on inefficient operators and decrease the overall fuzzing efficiency.

One operator's efficiency varies with target programs. Each operator's efficiency is program-dependent, and it is unlikely or at least difficult to statically infer this dependency. Thus the optimal mutation scheduler has to make decisions per program, relying on each operator's runtime efficiency on the target program.

One operator's efficiency varies over time. A mutation operator that performs well on the current test cases may perform poorly on the following test cases in extreme cases. As aforementioned, the optimal mutation scheduler rely on operators' history efficiency to calculate the optimal probability

distribution to select operators. Due to the dynamic characteristic of operator efficiency, this probability calculation process should converge fast.

The scheduler incurs performance overhead. Mutation schedulers have impacts on the execution speed of fuzzers. Since the execution speed is one of the key factors affecting fuzzers' efficiency, a better mutation scheduler should have fewer computations, to avoid slowing down fuzzers.

Unbalanced data for machine learning. During fuzzing, the numbers of positive and negative samples are not balanced, e.g., a mutation operator could only generate interesting test cases with a small probability, which may affect the effectiveness of gradient descent algorithms and other machine learning algorithms [8, 9].

In this paper, we consider mutation scheduling as an optimization problem and propose a novel mutation scheduling scheme MOPT, aiming at solving the aforementioned issues and improving the fuzzing performance. Inspired by the well-known optimization algorithm *Particle Swarm Optimization (PSO)* [20], MOPT dynamically evaluates the efficiency of candidate mutation operators, and adjusts their selection probability towards the optimum distribution.

MOPT models each mutation operator as a particle moving along the probability space $[x_{min}, x_{max}]$, where x_{min} and x_{max} are the pre-defined minimal and maximal probability, respectively. Guided by the local best probability and global best probability, each particle (i.e., operator) moves towards its optimal selection probability, which could yield more good-quality test cases. Accordingly, the target of MOPT is to find an optimal selection probability distribution of operators by aggregating the probabilities found by the particles, such that the aggregation yields more good-quality test cases. Similar to PSO, MOPT iteratively updates each particle's probability according to its local best probability and the global best probability. Then, it integrates the updated probabilities of all particles to obtain a new probability distribution. MOPT can quickly converge to the best solution of the probability distribution for selecting mutation operators and thus improves the fuzzing performance significantly.

MOPT is a generic scheme that can be applied to a wide range of mutation-based fuzzers. We have applied it to several state-of-the-art fuzzers, including AFL [17], AFLFast [6] and VUzzer [7], and implement MOPT-AFL, -AFLFast and -VUzzer, respectively. In AFL and its descendants, we further design a special pacemaker fuzzing mode, which could further accelerate the convergence speed of MOPT.

We evaluated these prototypes on 13 real world programs. In total, MOPT-AFL discovered 112 security vulnerabilities, including 97 previously unknown vulnerabilities (among which 66 are confirmed by CVE) and 15 known CVE vulnerabilities. Compared to AFL, MOPT-AFL found 170% more vulnerabilities, 350% more crashes and 100% more program paths. MOPT-*AFLFast* and MOPT-*VUzzer* also outperformed their counterparts on our dataset. We further demonstrated the rationality, steadiness and low costs of MOPT.

In summary, we have made the following contributions:

- We investigated the drawbacks of existing mutation schedulers, from which we conclude that mutation operators should be scheduled based on their history performance.
- We proposed a novel mutation scheduling scheme MOPT, which is able to choose better mutation operators and achieve better fuzzing efficiency. It can be generally applied to a broad range of existing mutation-based fuzzers.
- We applied MOPT to several state-of-the-art fuzzers, including AFL, AFLFast and VUzzer, and evaluated them on 13 real world programs. The results showed that MOPT could find much more vulnerabilities, crashes and program paths, with good steadiness, compatibility and low cost.
- MOPT-AFL discovers 97 previously unknown security vulnerabilities, and helps the vendors improve their products' security. It also finds 15 previously known vulnerabilities in these programs (of latest versions), indicating that security patching takes a long time in practice. We also open source MOPT-AFL along with the employed data, seed sets, and results at *https://github.com/puppet-meteor/MOpt-AFL* to facilitate the research in this area.

2 Background

2.1 Mutation-based Fuzzing

Mutation-based fuzzing [2, 6, 7, 14, 15, 16, 17, 18, 19] is good at discovering vulnerabilities, without utilizing prior knowledge (e.g., test case specification) of target programs. Instead, it generates new test cases by mutating some well-formed seed test cases in certain ways.

The general workflow of mutation-based fuzzing is as follows. The fuzzer (1) maintains a queue of seed test cases, which can be updated at runtime; (2) selects some seeds from the queue in certain order; (3) mutates the seeds in various ways; (4) tests target programs with the newly generated test cases, and reports vulnerabilities or updates the seed queue if necessary; then (5) goes back to step (2).

In order to efficiently guide the mutation and fuzzing, some fuzzers will also instrument target programs to collect runtime information during testing, and use it to guide seeds updating and decide which seeds to select and how to mutate them. In this paper, we mainly focus on the mutation phase (i.e., step (3)).

2.2 Mutation Operators

Mutation-based fuzzers could mutate seeds in infinite number of ways. Considering the performance and usability, in practice these fuzzers, including AFL [17] and its descendants, libFuzzer [18], honggfuzz [19], BFF [2] and VUzzer [7], usually predefine a set of mutation operators, and choose some of them to mutate seeds at runtime. These

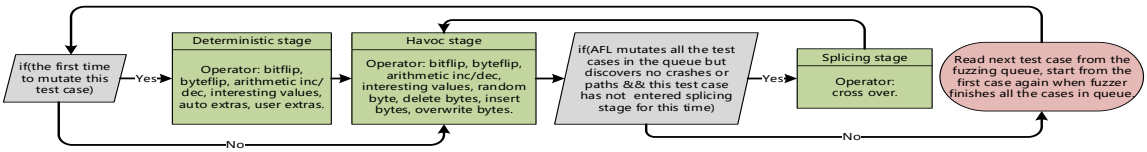

Figure 1: Three mutation scheduling schemes used in the three stages of AFL [17].

Table 1: Mutation operators defined by AFL [17].

Type	Meaning	Operators
bitflip	Invert one or several consecutive bits in a test case, where the stepover is 1 bit.	`bitflip 1/1`, `bitflip 2/1`, `bitflip 4/1`
byteflip	Invert one or several consecutive bytes in a test case, where the stepover is 8 bits.	`bitflip 8/8`, `bitflip 16/8`, `bitflip 32/8`
arithmetic inc/dec	Perform addition and subtraction operations on one byte or several consecutive bytes.	`arith 8/8`, `arith 16/8`, `arith 32/8`
interesting values	Replace bytes in the test cases with hard-coded interesting values.	`interest 8/8`, `interest 16/8`, `interest 32/8`
user extras	Overwrite or insert bytes in the test cases with user-provided tokens.	`user (over)`, `user (insert)`
auto extras	Overwrite bytes in the test cases with tokens recognized by AFL during `bitflip 1/1`.	`auto extras (over)`
random bytes	Randomly select one byte of the test case and set the byte to a random value.	`random byte`
delete bytes	Randomly select several consecutive bytes and delete them.	`delete bytes`
insert bytes	Randomly copy some bytes from a test case and insert them to another location in this test case.	`insert bytes`
overwrite bytes	Randomly overwrite several consecutive bytes in a test case.	`overwrite bytes`
cross over	Splice two parts from two different test cases to form a new test case.	`cross over`

mutation operators characterize where to mutate (e.g., which bytes) and how to mutate (e.g., add, delete or replace bytes).

For example, the well-recognized fuzzer AFL predefines 11 types of mutation operators, as shown in Table 1. In each type, there could be several concrete mutation operators. For instance, the `bitflip 2/1` operator flips 2 consecutive bits, where the stepover is 1 bit. Note that, different fuzzers could define different mutation operators.

2.3 Mutation Scheduling Schemes

At runtime, mutation-based fuzzers continuously select some predefined mutation operators to mutate seed test cases. *Different fuzzers have different schemes to select operators.* For example, AFL employs three different scheduling schemes used in three stages, as shown in Fig. 1.

1. **Deterministic stage scheduler.** AFL applies a deterministic scheduling scheme for seed test cases that are picked to mutate for the first time. This scheduler employs 6 deterministic types of mutation operators in order, and applies them on the seed test cases one by one. For instance, it will apply `bitflip 8/8` to flip each byte of the seed test cases.

2. **Havoc stage scheduler.** The major mutation scheduling scheme of AFL is used in the havoc stage. As shown in Fig. 2, AFL first decides the number, denoted as R_t, of new test cases to generate in this stage. Each time, AFL selects

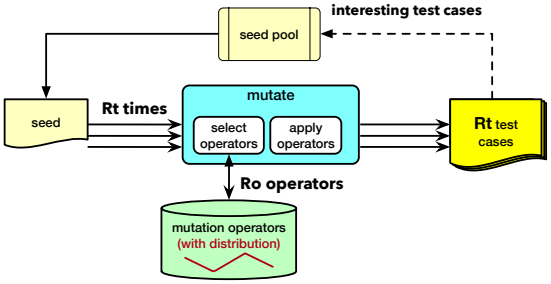

Figure 2: The general workflow of mutation-based fuzzing and mutation scheduling.

a series of R_o mutation operators following the uniform distribution, and applies them on the seed to generate one test case. The havoc stage ends after R_t new test cases have been generated.

3. **Splicing stage scheduler.** In some rare cases, AFL works through the aforementioned two stages for all seeds, but fails to discover any unique crash or path in one round. Then AFL will enter a special splicing stage. In this stage, AFL only employs one operator `cross over` to generate new test cases. These new test cases will be fed to the havoc stage scheduler, rather than the program being tested, to generate new test cases.

The mutation scheduler in the first stage is deterministic and slow, while the one in the last stage is rarely used. The scheduler in the havoc stage, as shown in Fig. 2, is more generic and has been widely adopted by many fuzzers. Therefore, in this paper we mainly focus on improving the scheduler used in the havoc stage, which thus can be implemented in most mutation-based fuzzers. More specifically, we aim at finding an optimal probability distribution, following which the scheduler could select better mutation operators and improve the fuzzing efficiency.

2.4 Mutation Efficiency

Different mutation operators work quite differently. An intuitive assumption is that, they have different efficiency on different target programs. Some are better than others at generating the test cases, denoted as *interesting test cases*, that can trigger new paths or crashes.

To verify our hypothesis, we conducted an experiment on AFL to evaluate each operator's efficiency. To make the evaluation result deterministic, we only measured the interesting test cases produced by 12 mutation operators in the deterministic stage. The result is demonstrated in Fig. 3.

In the deterministic stage, the order of mutation operators

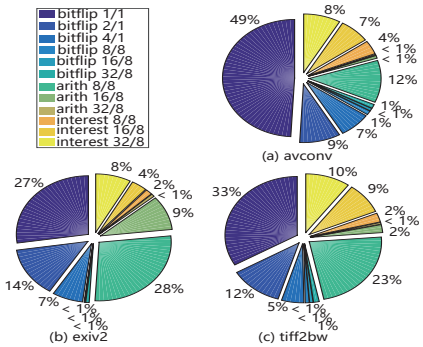

Figure 3: Percentages of interesting test cases produced by different operators in the deterministic stage of AFL.

and the times they are selected are fixed. Fig. 4 shows the order and the times that operators are selected by AFL during fuzzing `avconv`, indicating the time the fuzzer spent on.

• *Different mutation operators' efficiencies on one target program are different.* For most programs, the operators `bitflip 1/1`, `bitflip 2/1` and `arith 8/8` could yield more interesting test cases than other operators. On the other hand, several other mutation operators, such as `bitflip 16/8`, `bitflip 32/8` and `arith 32/8`, could only produce less than 2% of interesting test cases.

• *Each operator's efficiency varies with target programs.* An operator could yield good outputs on one program, but fail on another one. For example, `arith 8/8` performs well on `exiv2` and `tiff2bw`, but only finds 12% of the interesting test cases on `avconv`.

• *AFL spends most time on the deterministic stage.* We record the time each stage spends and the number of interesting test cases found by each stage in 24 hours, as shown in Fig. 5. We first analyze a special case. For `tiff2bw`, since AFL cannot find more interesting test cases, it finishes the deterministic stage of all the inputs in the fuzzing queue and skips the deterministic stage for a long time. Then, AFL spends most time on the havoc stage while finding nothing. For the other three cases, AFL spends more than 70% of the time on the deterministic stage. When fuzzing `avconv`, AFL even does not finish the deterministic stage of the first input in 24 hours. Another important observation is that the havoc stage is more efficient in finding interesting test cases compared to the deterministic stage. Moreover, since AFL spends too much time on the deterministic stage of one input, it cannot generate test cases from the later inputs in the fuzzing queue when fuzzing `avconv` and `pdfimages` given 24 hours. Note that since the splicing stage only uses `cross over` to mutate the test cases, it spends too little time to be shown in Fig. 5 compared to the other stages that will test the target program as mentioned in Section 2.3.

• *AFL spends much time on the inefficient mutation operators.* Fig. 3 shows that, the mutation operators `bitflip 1/1` and `bitflip 2/1` have found the most interesting test cases. But according to Fig. 4, they are only selected for

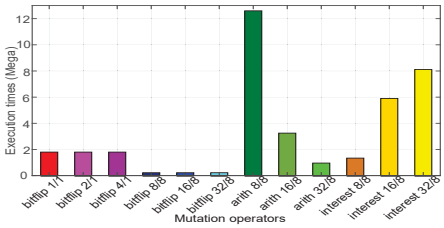

Figure 4: The times that mutation operators are selected when AFL fuzzes a target program `avconv`.

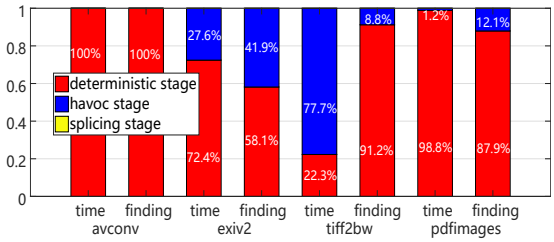

Figure 5: Percentages of time and interesting test cases used and found by the three stages in AFL, respectively.

a small number of times. On the other hand, inefficient operators like the ones of `interesting values` are selected too frequently but produce few interesting test cases, which decreases the fuzzing efficiency.

Motivation. Based on the analysis above, we observe that different mutation operators have different efficiencies. Hence, the mutation schedulers in existing fuzzers, which follow some pre-defined distributions, are not efficient. Ideally, more time should be spent on mutation operators that perform better at generating interesting test cases. Therefore, a better mutation scheduler is demanded.

3 Overview of MOPT

3.1 Design Philosophy

The mutation scheduler aims at choosing the next optimal mutation operator, which could find more interesting test cases, for a given runtime context. We simplify this problem as finding an optimal probability distribution of mutation operators, following which the scheduler chooses next operators when testing a target program.

Finding an optimal probability distribution for all mutation operators is challenging. Instead, we could first let each operator explore its own optimal probability. Then, based on those optimal probabilities, we could obtain a global optimal probability distribution of mutation operators.

The *Particle Swarm Optimization (PSO)* algorithm can be leveraged to find the optimal distribution of the operators and we detail the modification of PSO in our setting as follows.

3.2 Particle Swarm Optimization (PSO)

The PSO [20] algorithm is proposed by Eberhart and Kennedy, aiming at finding the optimal solution for a problem. It employs multiple particles to search the solution space iteratively, in which a position is a candidate solution.

As shown in Fig. 6, in each iteration, each particle is moved to a new position x_{now}, based on (1) its inertia (i.e., previous movement v_{now}), (2) displacement to its local best position L_{best} that *this particle* has found so far, and (3) displacement to the global best position G_{best} that *all particles* have found so far. Specifically, the movement of a particle P is calculated as follows:

$$v_{now}(P) \leftarrow w \times v_{now}(P) + r \times (L_{best}(P) - x_{now}(P)) + r \times (G_{best} - x_{now}(P)). \quad (1)$$

$$x_{now}(P) \leftarrow x_{now}(P) + v_{now}(P). \quad (2)$$

where w is the inertia weight and $r \in (0,1)$ is a random displacement weight.

Hence, each particle moves towards L_{best} and G_{best}, and is likely to keep moving to better positions. By moving towards G_{best}, multiple particles could work synchronously and avoid plunging into the local optimum. As a result, the swarm will be led to the optimal solution. Moreover, PSO is easy to implement with low computational cost, making it a good fit for optimizing mutation scheduling.

3.3 Design Details

MOPT aims to find an optimal probability distribution. Rather than employing particles to explore candidate distributions directly, we propose a customized PSO algorithm to explore each operator's optimal probability first, and then construct the optimal probability distribution.

3.3.1 Particles

MOPT employs a particle per operator, and tries to explore an optimal position for each operator in a predefined probability space $[x_{min}, x_{max}]$, where $0 < x_{min} < x_{max} \leq 1$.

The current position of a particle (i.e., operator) in the probability space, i.e., x_{now}, represents the probability that this operator will be selected by the scheduler. Due to the nature of probabilities, the sum of all the particles' probabilities in one iteration should be normalized to 1.

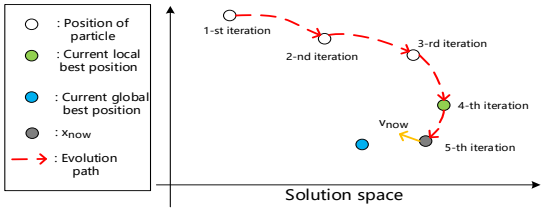

Figure 6: An example of illustrating the evolution of one particle at the 5-th iteration according to the PSO.

3.3.2 Local Best Position L_{best}

Similar to PSO, MOPT also appoints the best position that a particle has ever found as its local best position.

For a given particle, a position x_1 is better than x_2, if and only if, its corresponding operator yields more interesting test cases (with a same amount of invocations) in the former position than the latter. Thus, L_{best} is the position of the particle where the corresponding operator yields the most interesting test cases (given the same amount of invocations).

To enable this comparison, for each particle (i.e., operator), we measure its **local efficiency** eff_{now}, i.e., the number of interesting test cases contributed by this operator divided by the number of invocations of this operator *during one iteration*. We denote the largest eff_{now} as eff_{best}. Thus, L_{best} is the position where the operator obtains eff_{best} in history.

3.3.3 Global Best Position G_{best}

PSO appoints the best position that all particles have ever found as the global best position. Note that, unlike the original PSO which moves particles in a unified solution space, MOPT moves particles in different probability spaces (with same shape and size). Hence, there is no sole global best position fit for all particles. Instead, different particles have different global best positions (in different spaces) here.

In PSO, global best positions depend on the relationship between different particles. Hereby we also evaluate each particle's efficiency from a global perspective, denoted as **global efficiency** $global_{eff}$, by evaluating multiple swarms of particles at a time.

More specifically, we measure the number of interesting test cases contributed by each operator till now in all swarms, and use it as the particle's global efficiency $global_{eff}$. Then we compute the distribution of all particles' global efficiency. For each operator (i.e., particle), its global best position G_{best} is defined as the proportion of its $global_{eff}$ in this distribution. With this distribution, particles (i.e., operators) with higher efficiency can get higher probability to be selected.

3.3.4 Multiple Swarms

Given the definitions of particles, local best positions and global best positions, we could follow the PSO algorithm to approach to an optimal solution (i.e., a specific probability distribution of mutation operators).

However, unlike the original PSO swarm that has multiple particles exploring the solution space, the swarm defined by MOPT actually only explores one candidate solution (i.e., probability distribution) in the solution space, and thus is

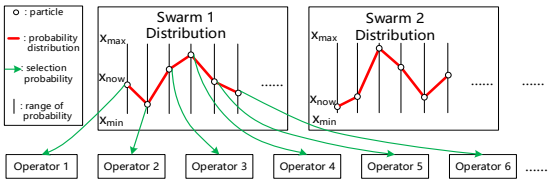

Figure 7: Illustration of the particle swarms of MOPT.

likely to fall into local optimum. Thus, MOPT employs multiple swarms and applies the customized PSO algorithm to each swarm, as shown in Fig. 7, to avoid local optimum.

Synchronization is required between these swarms. MOPT simply takes the most efficient swarm as the best and uses its distribution to schedule mutation during fuzzing. Here, we define the **swarm's efficiency** (denoted as $swarm_{eff}$) as the number of interesting test cases contributed by this swarm divided by the number of new test cases *during one iteration*.

Overview: In summary, MOPT employs multiple swarms and applies the customized PSO algorithm to each swarm. During fuzzing, the following three extra tasks are performed in each iteration of PSO.

• **T1:** *Locate local best positions for all particles in each swarm.* Within each swarm, each particle's local efficiency eff_{now} in one iteration is evaluated during fuzzing. For each particle, the position with the highest efficiency eff_{best} in history is marked as its local best position L_{best}.

• **T2:** *Locate global best positions for all particles across swarms.* Each particle's global efficiency $global_{eff}$ is evaluated across swarms. The distribution of the particles' global efficiency is then evaluated. The proportion of each particle's $global_{eff}$ in this distribution is used as its global best position G_{best}.

• **T3:** *Select the best swarm to guide fuzzing.* Each swarm's efficiency $swarm_{eff}$ in one iteration is evaluated. The swarm with the highest $swarm_{eff}$ is chosen, and its probability distribution in the current iteration is applied for further fuzzing.

Then, at the end of each iteration, MOPT moves the particles in each swarm in a similar way as PSO. More specifically, for a particle P_j in a swarm S_i, we update its position as follows.

$$\begin{aligned} v_{now}[S_i][P_j] \leftarrow & w \times v_{now}[S_i][P_j] \\ & + r \times (L_{best}[S_i][P_j] - x_{now}[S_i][P_j]) \\ & + r \times (G_{best}[P_j] \quad - x_{now}[S_i][P_j]). \end{aligned} \tag{3}$$

$$x_{now}[S_i][P_j] \leftarrow x_{now}[S_i][P_j] + v_{now}[S_i][P_j]. \tag{4}$$

where w is the inertia weight and $r \in (0,1)$ is a random displacement weight.

Further, we normalize these positions to meet some constraints. First, each particle's position is adjusted to fit in the probability space $[x_{min}, x_{max}]$. Then for each swarm, all its particles' positions (i.e., probabilities) will be normalized, such that the sum of these probabilities equals to 1.

After updating the positions of all particles in all swarms, the fuzzer could keep moving those particles into new positions, and enter a new iteration of PSO.

4 Implementation of MOPT

4.1 MOPT Main Framework

As shown in Fig. 8, MOPT consists of four core modules, i.e., the PSO initialization and updating modules, as well as the pilot fuzzing and core fuzzing modules.

The PSO initialization module is executed once and used for setting the initial parameters of the PSO algorithm. The other three modules form an iteration loop and work together to continuously fuzz target programs.

In each iteration of the loop, the PSO particles are updated once. In order to update particles' positions with the PSO algorithm, we need to find each particle's local best position and global best position in each iteration.

• The pilot fuzzing module employs multiple swarms, i.e., multiple probability distributions, to select mutation operators and fuzz. During fuzzing, the local efficiency of each particle in each swarm is measured. Hence, we could find the *local best position* of each particle in each swarm.

• Moreover, during the pilot fuzzing, each swarm's efficiency is also evaluated. Then, the *most efficient swarm* is chosen, and the core fuzzing module will use the probability distribution explored by it to schedule mutation operators.

• After the core fuzzing module finishes, the total number of interesting test cases contributed by each operator till now can be evaluated. Hence, each particle's global efficiency (i.e., *global best position*) could be evaluated.

With this iteration loop, the fuzzer could utilize the PSO to find an optimal probability distribution to select mutation operators, and gradually improve the fuzzing efficiency.

Note that, MOPT's workflow is independent from the target fuzzer, as long as the fuzzer's mutation scheduler uses a probability distribution to select operators. We do not need to change the behavior of the target fuzzer, except that evaluating the efficiency of the fuzzer in order to move PSO particles. The instrumentation to the target fuzzer is minimum and costs few performance overhead.

Hence, MOPT is a generic and practical mutation scheduling scheme, and can be applied to a variety of fuzzers.

4.1.1 PSO Initialization Module

This module initializes parameters for the PSO algorithm. More specifically, MOPT (1) sets the initial location x_{now} of each particle in each swarm with a random value, and normalizes the sum of x_{now} of all the particles in one swarm to 1; (2) sets the displacement of particle movement v_{now} of each particle in each swarm to 0.1; (3) sets the initial local efficiency eff_{now} of each particle in each swarm to 0;

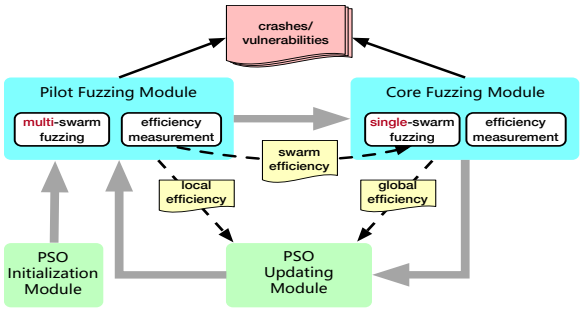

Figure 8: The workflow of MOPT.

(4) sets the initial local best position L_{best} of each particle in each swarm to 0.5; and (5) sets the initial global best position G_{best} of each particle across swarms to 0.5. Note that, the initialization module only executes once when the fuzzer starts running.

4.1.2 Pilot Fuzzing Module

This module employs multiple swarms to perform fuzzing, where each swarm explores a different probability distribution. This module evaluates each swarm in order, and stops testing a swarm after it has generated a configurable number (denoted as $period_{pilot}$) of new test cases. The process of fuzzing with a specific swarm is as follows.

For each swarm, its probability distribution is used to schedule the selection of mutation operators and fuzz the target program. During fuzzing, the module will measure three measurements: (1) the number of interesting test cases contributed by a specific particle (i.e., operator), (2) the number of invocations of a specific particle, (3) the number of interesting test cases found by this swarm, by instrumenting target programs.

The local efficiency of each particle (in current swarm) is the first measurement divided by the second measurement. Hence, we could locate the *local best position* of each particle. The current *swarm's efficiency* is the third measurement divided by the test case count $period_{pilot}$. Therefore, we could find the most efficient swarm.

4.1.3 Core Fuzzing Module

This module will take the best swarm selected by the pilot fuzzing module, and use its probability distribution to perform fuzzing. It will stop after generating a configurable number (denoted as $period_{core}$) of new test cases.

Once it stops, we could measure the number of interesting test cases contributed by each particle, regardless which swarm it belongs to, from the start of PSO initialization till now. Then we could calculate the distribution between particles, and locate each particle's *global best position*.

Note that, if we only use one swarm in the pilot module, then the core module could be merged with the pilot module.

4.1.4 PSO Updating Module

With the information provided by the pilot and core fuzzing modules, this module updates the particles in each swarm, following Equations 3 and 4.

After updating each particle, we will enter the next iteration of PSO updates. Hence, we could approach to an optimal swarm (i.e., probability distribution for operators), use it to guide the core fuzzing module, and help improve the fuzzing efficiency.

4.2 Pacemaker Fuzzing Mode

Although applying MOPT to mutation-based fuzzers is generic, we realize the performance of MOPT can be further optimized when applied to specific fuzzers such as AFL.

Based on extensive empirical analysis, we realize that AFL and its descendants spend much more time on the deterministic stage, than on the havoc and splicing stages that can discover many more unique crashes and paths. MOPT therefore provides an optimization to AFL-based fuzzers, denoted as *pacemaker fuzzing mode*, which selectively avoids the time-consuming deterministic stage.

Specifically, when MOPT finishes mutating one seed test case, if it has not discovered any new unique crash or path for a long time, i.e., T that is set by users, it will selectively disable the deterministic stage for the following test case. The pacemaker fuzzing mode has the following advantages.

• The deterministic stage spends too much time and would slow down the overall efficiency. On the other hand, MOPT only updates the probability distribution in the havoc stage, independent from the deterministic stage. Therefore, disabling the deterministic stage with *the pacemaker fuzzing mode* could accelerate the convergence speed of MOPT.

• In this mode, the fuzzer can skip the deterministic stage, without spending too much time on a sole test case. Instead, it will pick more seeds from the fuzzing queue for mutation, and thus has a better chance to find vulnerabilities faster.

• The deterministic stage may have good performance at the beginning of fuzzing, but becomes inefficient after a while. This mode selectively disables this stage only after the efficiency slows down, and thus benefits from this stage while avoiding wasting much time on it.

More specifically, MOPT provides two types of pacemaker fuzzing modes for AFL, based on whether the deterministic stage will be re-enabled or not: (1) MOPT-*AFL-tmp*, which will re-enable the deterministic stage again when the number of new interesting test cases exceeds a predefined threshold; (2) MOPT-*AFL-ever*, which will never re-enable the deterministic stage in the following fuzzing process.

Table 2: Objective programs evaluated in our experiments.

Target	Source file	Input format	Test instruction
mp42aac	Bento4-1-5-1 [22]	mp4	mp42aac @@ /dev/null
exiv2	exiv2-0.26-trunk [23]	jpg	exiv2 @@ /dev/null
mp3gain	mp3gain-1_5_2 [24]	mp3	mp3gain @@ /dev/null
tiff2bw	libtiff-4.0.9 [25]	tiff	tiff2bw @@ /dev/null
pdfimages	xpdf-4.00 [26]	PDF	pdfimages @@ /dev/null
sam2p	sam2p-0.49.4 [27]	bmp	sam2p @@ EPS: /dev/null
avconv	libav-12.3 [28]	mp4	avconv -y -i @@ -f null -
w3m	w3m-0.5.3 [29]	text	w3m @@
objdump	binutils-2.30 [30]	binary	objdump –dwarf-check -C -g -f -dwarf -x @@
jhead	jhead-3.00 [31]	jpg	jhead @@
mpg321	mpg321_0.3.2 [32]	mp3	mpg321 -t @@ /dev/null
infotocap	ncurses-6.1 [33]	text	infotocap @@
podofopdfinfo	podofo-0.9.6 [34]	PDF	podofopdfinfo @@

5 Evaluation

5.1 Real World Datasets

We have evaluated MOPT on 13 open-source linux programs as shown in Table 2, each of which comes from different source files and has different functionality representing a broad range of programs. We choose these 13 programs mainly for the following reasons. First, many of the employed programs are also widely used in state-of-the-art fuzzing research [5, 6, 10, 11, 21]. Second, most programs employed in our experiments are real world programs from different vendors and have diverse functionalities and various code logic. Therefore, our datasets are representative and can all-sidedly measure the fuzzing performance of fuzzers to make our analysis more comprehensive. Third, all the employed programs are popular and useful open-source programs. Hence, evaluating the security of these programs are meaningful for the vendors and users of them.

5.2 Experiment Settings

The version of AFL used in our paper is 2.52b. We apply MOPT in the havoc stage of AFL and implement the prototypes of MOPT-AFL-tmp and MOPT-AFL-ever, where -tmp and -ever indicate the corresponding pacemaker fuzzing modes discussed in the previous section. The core functions of MOPT is implemented in C.

Platform. All the experiments run on a virtual machine configured with 1 CPU core of 2.40GHz E5-2640 V4, 4.5GB RAM and the OS of 64-bit Ubuntu 16.04 LTS.

Initial seed sets. Following the same seed collection and selection procedure as in [4, 35, 36], we use randomly-selected files as the initial seed sets. In particular, for each objective program, we obtain 100 files with the corresponding input format as the initial seed set, e.g., we collect 100 mp3 files for `mp3gain`. The input format of each program is shown in Table 2. In particular, we first download the files with the corresponding input formats for each objective program from the music download websites, picture download websites, and so on (except for text files, where we obtain text files by randomly generating letters to fill them). Then, for the large files such as mp3 and PDF, we split them to make their sizes reasonable as seeds. Through this way, we have a large corpus of files with the corresponding input formats for each objective program. Finally, we randomly select 100 files from the corpus. These 100 files will be the initial seed set of all the fuzzers when fuzzing one objective program.

Evaluation metrics. The main evaluation metric is the number of the unique crashes discovered by each fuzzer. Since coverage-based fuzzers such as *AFLFast* [6] and *VUzzer* [7] consider that exploring more unique paths leads to more unique crashes, the second evaluation metric is the number of unique paths discovered by each fuzzer.

5.3 Unique Crashes and Paths Discovery

We evaluate AFL, MOPT-AFL-tmp and MOPT-AFL-ever on the 13 programs in Table 2, with each experiment runs for 240 hours. The results are shown in Table 3. To demonstrate the steadiness of MOPT, we repeat the experiments to avoid the potential randomness and more details can be found in Section 6.1.

We can deduce the following conclusions from Table 3.

• For discovering unique crashes, MOPT-AFL-tmp and MOPT-AFL-ever are significantly more efficient than AFL on all the programs. For instance, MOPT-AFL-ever finds 600 more unique crashes than AFL on `infotocap`. MOPT-AFL-tmp discovers 506 more unique crashes than AFL on `w3m`. In total, MOPT-AFL-tmp and MOPT-AFL-ever discover 2,195 and 2,334 more unique crashes than AFL on the 13 programs. Therefore, MOPT-AFL has much better performance than AFL in exploring unique crashes.

• For triggering unique paths, MOPT-AFL-tmp and MOPT-AFL-ever also significantly outperform AFL. For instance, MOPT-AFL-tmp discovers 9,746 more unique paths than AFL on `pdfimages`. In total, MOPT-AFL-tmp and MOPT-AFL-ever found 45,600 and 56,515 more unique paths than AFL on the 13 programs. As a result, the proposed MOPT can improve the coverage of AFL remarkably.

• When considering the pacemaker fuzzing mode, MOPT-AFL-tmp and MOPT-AFL-ever discover the most unique crashes on 8 and 6 programs respectively, while MOPT-AFL-ever discovers more crashes in total. Since the main difference between the two fuzzers is whether using the deterministic stage later, it may be an interesting future work to figure out how to employ the deterministic stage properly.

In summary, *both* MOPT-*AFL-tmp and* MOPT-*AFL-ever are much more efficient in finding unique crashes and paths than AFL, while their performance are comparable.*

Table 3: The unique crashes and paths found by AFL, MOPT-AFL-tmp and MOPT-AFL-ever on the 13 real world programs.

Program	AFL		MOPT-AFL-tmp				MOPT-AFL-ever			
	Unique crashes	Unique paths	Unique crashes	Increase	Unique paths	Increase	Unique crashes	Increase	Unique paths	Increase
mp42aac	135	815	**209**	+54.8%	1,660	+103.7%	199	+47.4%	**1,730**	+112.3%
exiv2	34	2,195	54	+58.8%	2,980	+35.8%	**66**	+94.1%	**4,642**	+111.5%
mp3gain	178	1,430	**262**	+47.2%	**2,211**	+54.6%	**262**	+47.2%	2,206	+54.3%
tiff2bw	4	4,738	**85**	+2,025.0%	**7,354**	+55.2%	43	+975.0%	7,295	+54.0%
pdfimages	23	12,915	357	+1,452.2%	22,661	+75.5%	**471**	+1,947.8%	**26,669**	+106.5%
sam2p	36	531	105	+191.7%	1,967	+270.4%	**329**	+813.9%	**3,418**	+543.7%
avconv	0	2,478	**4**	+4	**17,359**	+600.5%	1	+1	16,812	+578.5%
w3m	0	3,243	**506**	+506	5,313	+63.8%	182	+182	**5,326**	+64.2%
objdump	0	11,565	**470**	+470	19,309	+67.0%	287	+287	**22,648**	+95.8%
jhead	19	478	55	+189.5%	**489**	+2.3%	**69**	+263.2%	483	+1.0%
mpg321	10	123	**236**	+2,260.0%	1,054	+756.9%	229	+2,190.0%	**1,162**	+844.7%
infotocap	92	3,710	340	+269.6%	6,157	+66.0%	**692**	+652.2%	**7,048**	+90.0%
podofopdfinfo	79	3,397	**122**	+54.4%	**4,704**	+38.5%	114	+44.3%	4,694	+38.2%
total	610	47,618	2,805	+359.8%	93,218	+95.8%	**2,944**	+382.6%	**104,133**	+118.7%

Table 4: Vulnerabilities found by AFL, MOPT-AFL-tmp and MOPT-AFL-ever.

Program	AFL				MOPT-AFL-tmp				MOPT-AFL-ever			
	Unknown vulnerabilities		Known vulnerabilities	Sum	Unknown vulnerabilities		Known vulnerabilities	Sum	Unknown vulnerabilities		Known vulnerabilities	Sum
	Not CVE	CVE	CVE		Not CVE	CVE	CVE		Not CVE	CVE	CVE	
mp42aac	/	1	1	2	/	2	1	3	/	5	1	6
exiv2	/	5	3	8	/	5	4	9	/	4	4	8
mp3gain	/	4	2	6	/	9	3	12	/	5	2	7
pdfimages	/	1	0	1	/	12	3	15	/	9	2	11
avconv	/	0	0	0	/	2	0	2	/	1	0	1
w3m	/	0	0	0	/	14	0	14	/	5	0	5
objdump	/	0	0	0	/	1	2	3	/	0	2	2
jhead	/	1	0	1	/	4	0	4	/	5	0	5
mpg321	/	0	1	1	/	0	1	1	/	0	1	1
infotocap	/	3	0	3	/	3	0	3	/	3	0	3
podofopdfinfo	/	5	0	5	/	6	0	6	/	6	0	6
tiff2bw	1	/	/	1	2	/	/	2	2	/	/	2
sam2p	5	/	/	5	14	/	/	14	28	/	/	28
Total	6	20	7	33	16	58	14	88	30	43	12	85

5.4 Vulnerability Discovery

To figure out the corresponding vulnerabilities of the crashes found in Section 5.3, we recompile the evaluated programs with *AddressSanitizer* [37] and reevaluate them with the discovered crash inputs. If the top three source code locations of the stack trace provided by *AddressSanitizer* are unique, we consider the corresponding crash input triggers a unique vulnerability of the objective program. This is a common way to find unique vulnerabilities in practice and has been used to calculate the stack hashing in [38]. Then, we check the vulnerability reports of the target program on the CVE website to see whether they correspond to some already existed CVEs. If not, we submit the vulnerability reports and the Proof of Concepts (PoCs) to the vendors and the CVE assignment team. The vulnerabilities discovered by AFL, MOPT-AFL-tmp and MOPT-AFL-ever are shown in Table 4, from which we have the following conclusions.

• Both MOPT-AFL-tmp and MOPT-AFL-ever discover more vulnerabilities than AFL by a wide margin. For instance, MOPT-AFL-tmp finds 45 more security CVEs than AFL; MOPT-AFL-ever finds 23 more unreported CVEs than AFL; Our fuzzers find 81 security CVEs with 66 new CVE IDs assigned on 11 programs. The results demonstrate that MOPT-AFL is very effective on exploring CVEs.

• In the experiments, our fuzzers discover 15 previously known vulnerabilities published by CVE on the latest version of the objective programs. For instance, when fuzzing `pdfimages`, MOPT-AFL-tmp and MOPT-AFL-ever discover 3 and 2 existed vulnerabilities respectively, and when fuzzing `exiv2`, both MOPT-AFL-tmp and MOPT-AFL-ever discover 4 existing vulnerabilities. The results demonstrate that security patching takes a long time in practice.

• AFL, MOPT-AFL-tmp and MOPT-AFL-ever discover 1, 2 and 2 unique vulnerabilities on `tiff2bw`, respectively. As for `sam2p`, MOPT-AFL-tmp and MOPT-AFL-ever discover 14 and 28 unique vulnerabilities, respectively. In comparison, AFL only finds 5 vulnerabilities. Since the vulnerabilities happened in the `tiff2bw` command-line program and the CVE assignment team thinks that `sam2p` is a UNIX command line program rather than a library, they cannot assign CVE IDs for the vulnerabilities on `tiff2bw` and `sam2p`. Thus, we only provide the vendors of `tiff2bw` and `sam2p` with the vulnerabilities. On all the 13 programs, MOPT-AFL-tmp and MOPT-AFL-ever discover 112 unique vulnerabilities in total, and AFL discovers 33 vulnerabilities.

Table 5: The types and IDs of CVE discovered by AFL, MOPT-AFL-tmp and MOPT-AFL-ever.

Target	Types	AFL	MOPT-AFL-tmp	MOPT-AFL-ever	Severity
mp42aac	buffer overflow	CVE-2018-10785	CVE-2018-10785; CVE-2018-18037	CVE-2018-10785; CVE-2018-18037; CVE-2018-17814	4.3
	memory leaks	CVE-2018-17813	CVE-2018-17813	CVE-2018-17813; CVE-2018-18050; CVE-2018-18051	4.3
exiv2	heap overflow	CVE-2017-11339; CVE-2017-17723; CVE-2018-18036	CVE-2017-11339; CVE-2017-17723; CVE-2018-10780	CVE-2017-11339; CVE-2017-17723; CVE-2018-18036	5.8
	stack overflow	CVE-2017-14861	CVE-2017-14861	CVE-2017-14861	4.3
	buffer overflow	CVE-2018-18047	CVE-2018-17808; CVE-2018-18047	CVE-2018-18047	4.3
	segmentation violation	CVE-2018-18046	CVE-2018-18046	CVE-2018-18046	4.3
	memory access violation	CVE-2018-17809; CVE-2018-17807	CVE-2018-17809; CVE-2018-17823	CVE-2017-11337; CVE-2018-17809	4.3
mp3gain	stack buffer overflow	CVE-2017-14407	CVE-2017-14407; CVE-2018-17801; CVE-2018-17799	CVE-2017-14407	4.3
	global buffer overflow	CVE-2018-17800; CVE-2018-17802; CVE-2018-18045; CVE-2018-18043	CVE-2017-14409; CVE-2018-17800; CVE-2018-17803; CVE-2018-17802; CVE-2018-18045; CVE-2018-18043; CVE-2018-18044	CVE-2018-17800; CVE-2018-17803; CVE-2018-17802; CVE-2018-18045; CVE-2018-18043	6.8
	segmentation violation	CVE-2017-14406	CVE-2017-14412	CVE-2017-14412	6.8
	memcpy param overlap		CVE-2018-17824		5.8
pdfimages	heap buffer overflow		CVE-2018-8103; CVE-2018-18054		4.3
	stack overflow	CVE-2018-17114	CVE-2018-16369; CVE-2018-17114; CVE-2018-17115; CVE-2018-17116; CVE-2018-17117; CVE-2018-17119; CVE-2018-17120; CVE-2018-17121; CVE-2018-17122; CVE-2018-18053; CVE-2018-18055	CVE-2018-16369; CVE-2018-17115; CVE-2018-17116; CVE-2018-17119; CVE-2018-17121; CVE-2018-17122; CVE-2018-18053	6.1
	global buffer overflow		CVE-2018-8102	CVE-2018-8102	4.3
	alloc dealloc mismatch		CVE-2018-17118	CVE-2018-17118	4.3
	segmentation violation			CVE-2018-17123; CVE-2018-17124	4.3
avconv	segmentation violation		CVE-2018-17804	CVE-2018-17804	4.3
	memory leaks		CVE-2018-17805		4.3
w3m	segmentation violation		CVE-2018-17815; CVE-2018-17816; CVE-2018-17817; CVE-2018-17818; CVE-2018-17819; CVE-2018-17821; CVE-2018-17822; CVE-2018-18038; CVE-2018-18039; CVE-2018-18040; CVE-2018-18041; CVE-2018-18042; CVE-2018-18052	CVE-2018-17816; CVE-2018-18040; CVE-2018-18041; CVE-2018-18042	5.3
	memory leaks		CVE-2018-17820	CVE-2018-17820	4.3
objdump	stack exhaustion		CVE-2018-12700	CVE-2018-12641	5.0
	stack overflow		CVE-2018-9138; CVE-2018-16617	CVE-2018-9138	4.3
jhead	heap buffer overflow	CVE-2018-17810	CVE-2018-17810; CVE-2018-17811; CVE-2018-18048; CVE-2018-18049	CVE-2018-17810; CVE-2018-17811; CVE-2018-17812; CVE-2018-18048; CVE-2018-18049	4.3
mpg321	heap buffer overflow	CVE-2017-12063	CVE-2017-12063	CVE-2017-12063	4.3
infotocap	memory leaks	CVE-2018-16614	CVE-2018-16614	CVE-2018-16614	4.3
	segmentation violation	CVE-2018-16615; CVE-2018-16616	CVE-2018-16615; CVE-2018-16616	CVE-2018-16615; CVE-2018-16616	4.3
podofopdfinfo	stack overflow	CVE-2018-18216; CVE-2018-18221; CVE-2018-18222	CVE-2018-18216; CVE-2018-18217; CVE-2018-18221; CVE-2018-18222	CVE-2018-18216; CVE-2018-18217; CVE-2018-18218; CVE-2018-18221	4.7
	heap buffer overflow	CVE-2018-18219	CVE-2018-18219	CVE-2018-18219	4.3
	segmentation violation	CVE-2018-18220	CVE-2018-18220	CVE-2018-18220	4.3

In summary, MOPT-*AFL has better performance on exploring vulnerabilities than AFL.* MOPT-*AFL-tmp and* MOPT-*AFL-ever achieve similar performance, discovering 88 and 85 vulnerabilities on 13 programs, respectively.*

5.5 CVE Analysis

In this subsection, we analyze the CVEs discovered in Section 5.4 in detail and discuss the performance of different fuzzers. We also measure the severity of each CVE for each program by leveraging the Common Vulnerability Scoring System (CVSS) [39] and show the highest score in Table 5. We can learn the following conclusions.

• Both MOPT-AFL-tmp and MOPT-AFL-ever find more kinds of vulnerabilities than AFL, which means MOPT-AFL does not limit on discovering specific kinds of vulnerabilities. In other words, the MOPT scheme can guide the fuzzing tools to discover various vulnerabilities.

• We realize that MOPT-AFL-tmp discovers significantly more unique vulnerabilities than MOPT-AFL-ever on `pdfimages` and `w3m`. We analyze the reasons as follows. First of all, we would like to clarify the functionalities of these two objective programs. `pdfimages` is used to save images from the PDF files as the image files locally. `w3m` is a pager and/or text-based browser, which can handle tables, cookies, authentication, and almost everything except for JavaScript. We notice that PDF files have complex structures and so do the web data handled by `w3m`. Thus, there are many magic byte checks in `pdfimages` and `w3m` to handle the complex structures. Because it is hard to randomly generate a particular value, the operators in the deterministic stage, such as flipping the bits one by one (**bitflip**) and replacing the bytes with interesting values (**interesting values**), are better than the ones in the havoc stage to pass the magic byte checks and to test deeper execution paths. MOPT-AFL-tmp performs better than MOPT-AFL-ever on `pdfimages` and `w3m` since MOPT-AFL-tmp enables the deterministic stage later while MOPT-AFL-ever does not. However, since the deterministic stage performs multiple kinds of operators on each bit/byte of the test cases, it takes a lot of time to finish all the operations on each test case in the fuzzing queue, leading to the low efficiency. On the other hand, MOPT-AFL-tmp temporarily uses the deterministic stage on different test cases in the fuzzing queue to avoid this disadvantage.

• Interestingly, we can also see that although we fuzz the objective programs with the latest version, MOPT-AFL still discovers already existed CVEs. For instance, we reproduce the Proof of Concepts (PoCs) of CVE-2017-17723 of `exiv2`, which can cause the overflow and has 5.8 CVSS Score according to *CVE Details* [40]. It may because the vendors do not patch the vulnerabilities before the release or they patch the vulnerabilities while MOPT-AFL still discovers other PoCs. Therefore, the servers using these programs may be attacked because of these vulnerabilities. In addition, most of the discovered vulnerabilities can crash the programs and allow remote attackers to launch denial of service attacks via a crafted file. Thus, a powerful fuzzer is needed to improve the security patching.

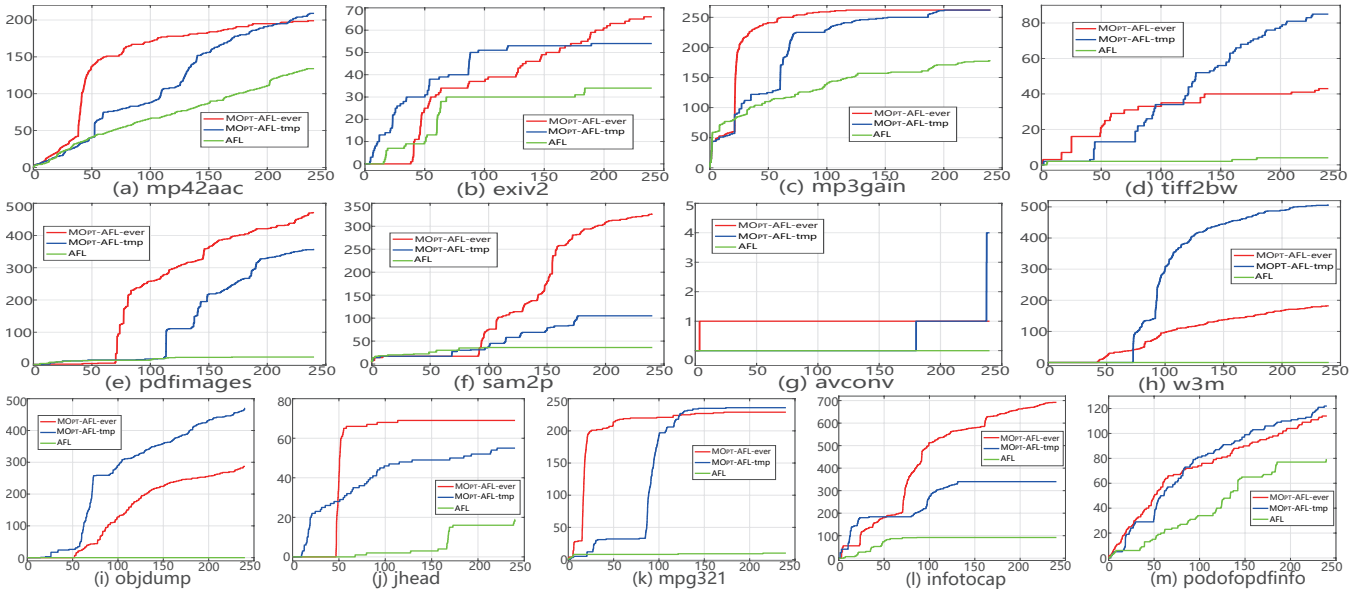

Figure 9: The number of unique crashes discovered by MOPT-AFL-ever, MOPT-AFL-tmp and AFL over 240 hours. X-axis: time (over 240 hours). Y-axis: the number of unique crashes.

Case study: CVE-2018-18054 in `pdfimages`. An interesting vulnerability we found is a heap buffer overflow in `pdfimages`. Although the PDF files in the seed set do not contain pictures that use the CCITTFax encoding, a test case generated by MOPT-AFL-tmp still triggers the CCITTFax decoding process of `pdfimages`. Furthermore, even the PDF syntax of this test case is partially damaged, `pdfimages` continues to extract the pictures from it. Then, the test case triggers the function `GBool CCITTFaxStream::readRow()` in `Stream.cc` for multiple times and finally accesses the data that exceed the index of the array `refLine`, which leads to a heap buffer overflow. This vulnerability shows the powerful mutation capability of MOPT-AFL-tmp, which not only generates a structure similar to an encoding algorithm but also triggers an array out of bounds.

5.6 More Analysis on Discovered Crashes

In this subsection, we give a close look on the growth of the number of unique crashes discovered by MOPT-AFL-ever, MOPT-AFL-tmp and AFL. The results are shown in Fig. 9, from which we have the following conclusions.

• Both MOPT-AFL-ever and MOPT-AFL-tmp are effective at finding unique crashes. On most programs, they take fewer than 100 hours to find more unique crashes than AFL does in 240 hours.

• We can learn from Fig. 9 (c) and (f) that within a relatively short time, AFL may discover more crashes than MOPT-AFL-ever and MOPT-AFL-tmp. The reasons are as follows. First, mutation-based fuzzing has randomness, and naturally such randomness may cause performance shaking within a short time-scale. However, relatively stable performance will exhibit in a long time-scale as shown in the experiments. Second, the selection probability distribution of mutation operators in MOPT-AFL-ever and MOPT-AFL-tmp adopts random initialization, which may cause fuzzing randomness in the early fuzzing time. Thus, to reduce the performance instability of fuzzing, a relatively long time experiment is necessary, e.g., we run our experiments for 240 hours.

5.7 Compatibility Analysis

In addition to AFL, we also generalize our analysis to several state-of-the-art mutation-based fuzzers, e.g., AFLFast [6] and VUzzer [7], and study the compatibility of MOPT.

AFLFast [6] is one of the coverage-based fuzzers. By using a power schedule to guide the fuzzer towards low frequency paths, AFLFast can detect more unique paths and explore the vulnerabilities much faster than AFL. To examine the compatibility of MOPT, we implement MOPT-AFLFast-tmp and MOPT-AFLFast-ever based on AFLFast.

VUzzer [7] is a fuzzer that focuses on exploring deeper paths, which is widely different from AFL. VUzzer can evaluate a test case with the triggered path and select the test cases with higher fitness scores to generate subsequent test cases. The mutation strategy of VUzzer is different from AFL. In each period, VUzzer generates a fixed number of mutated test cases, evaluates their fitness and only keeps *POPSIZE* test cases with the highest fitness scores to generate test cases in the next period, where *POPSIZE* is the population number of the parent test cases set by users. We regard the mutation operators as the high-efficiency operators that can generate the test cases with top-(*POPSIZE/3*) fitness scores. Then, we combine MOPT with VUzzer and implement MOPT-VUzzer. Since VUzzer does not have a

Table 6: The compatibility of the MOPT scheme.

		mp42aac	exiv2	mp3gain	tiff2bw	pdfimages	sam2p	mpg321
AFL	Unique crashes	135	34	178	4	23	36	10
	Unique paths	815	2,195	1,430	4,738	12,915	531	123
MOPT-AFL-tmp	Unique crashes	**209**	54	**262**	**85**	357	105	**236**
	Unique paths	1,660	2,980	**2,211**	**7,354**	22,661	1,967	1,054
MOPT-AFL-ever	Unique crashes	199	**66**	**262**	43	**471**	**329**	229
	Unique paths	**1,730**	**4,642**	2,206	7,295	**26,669**	**3,418**	**1,162**
AFLFast	Unique crashes	210	0	171	0	18	37	8
	Unique paths	1,233	159	1,383	5,114	12,022	603	122
MOPT-AFLFast-tmp	Unique crashes	**393**	51	**264**	5	292	**196**	**230**
	Unique paths	**3,389**	2,675	2,017	7,012	24,164	2,587	**1,208**
MOPT-AFLFast-ever	Unique crashes	384	**58**	259	**18**	**345**	114	30
	Unique paths	2,951	**2,887**	**2,102**	**7,642**	**26,799**	**2,623**	160
VUzzer	Unique crashes	12	0	54,500	0	0	13	3,598
	Unique paths	12%	9%	50%	13%	25%	18%	18%
MOPT-VUzzer	Unique crashes	**16**	0	**56,109**	0	0	**16**	**3,615**
	Unique paths	12%	9%	**51%**	13%	25%	18%	18%

deterministic stage like AFL, we do not consider the pacemaker fuzzing mode here.

Now, we evaluate MOPT-AFLFast-tmp, MOPT-AFLFast-ever, and MOPT-VUzzer on `mp42aac`, `exiv2`, `mp3gain`, `tiff2bw`, `pdfimages`, `sam2p` and `mpg321`. Each experiment is lasted for 240 hours with the same settings as in Section 5.2. Specifically, we change the OS as the 32-bit Ubuntu 14.04 LTS for VUzzer and MOPT-VUzzer because of VUzzer's implementation restriction. The results are shown in Table 6. We have the following conclusions.

• MOPT-AFLFast-tmp and MOPT-AFLFast-ever have much better performance than AFLFast in discovering unique crashes on all the programs. For instance, MOPT-AFLFast-tmp finds 183 more unique crashes than AFLFast on `mp42aac`; MOPT-AFLFast-ever finds 327 more crashes than AFLFast on `pdfimages`. When combining MOPT with VUzzer, MOPT-VUzzer discovers more unique crashes than VUzzer on `mp42aac`, `mp3gain`, `sam2p` and `mpg321`. As a result, MOPT cannot only be combined with state-of-the-art fuzzers like AFLFast, but also be compatible with the different fuzzers like VUzzer to improve the fuzzing performance.

• MOPT-based fuzzers can explore more unique paths than their counterparts. For instance, MOPT-AFLFast-tmp discovers 2,156 more paths than AFLFast on `mp42aac`; MOPT-AFLFast-ever finds 14,777 more than AFLFast on `pdfimages`. MOPT-VUzzer has a better coverage performance than VUzzer on `mp3gain`. Overall, MOPT can help the mutation-based fuzzers discover more unique paths.

• MOPT-AFL has an outstanding performance in comparison to state-of-the-art fuzzers. MOPT-AFL outperforms AFLFast with a significant advantage on all the programs except `mp42aac`. For instance, MOPT-AFL-tmp and MOPT-AFL-tmp discover 85 and 43 more unique crashes than AFLFast on `tiff2bw`. Furthermore, MOPT-AFL-tmp and MOPT-AFL-ever find dozens of times more unique crashes than VUzzer on most programs.

In summary, MOPT *is easily compatible with state-of-the-art mutation-based fuzzers even though they have different architectures.* More importantly, according to the results, MOPT-*AFL outperforms the state-of-the-art fuzzers such as AFLFast and VUzzer by a wide margin in many scenarios.*

5.8 Evaluation on LAVA-M

Recently, the LAVA-M dataset is proposed as one of the standard benchmarks to examine the performance of fuzzers [41]. It has 4 target programs, and each of which contains listed and unlisted bugs. The authors provided test cases that can trigger the listed bugs. However, no test cases were provided for the unlisted bugs, making them more difficult to be found. For completeness, we test AFL, MOPT-AFL-ever, AFLFast, MOPT-AFLFast-ever, VUzzer and MOPT-VUzzer on LAVA-M with the same initial seed set and the same settings as in Section 5.7, for 5 hours. Furthermore, we run MOPT-AFL-ever with Angora [10] and QSYM [42] parallelly to construct MOPT-Angora and MOPT-QSYM, run AFL with them parallelly to construct AFL-Angora and AFL-QSYM, and evaluate them on LAVA-M under the same experiment settings. The results are shown in Table 7, from which we have the following conclusions.

• MOPT-based fuzzers significantly outperform their counterparts on LAVA-M. For instance, MOPT-AFL-ever finds 35 more listed bugs than AFL on `base64`. MOPT-AFLFast-ever discovers 17 more listed bugs than AFLFast on `md5sum`. MOPT-VUzzer finds more listed bugs than VUzzer on all the four programs. Both MOPT-Angora and MOPT-QSYM find significantly more unique bugs on `who` compared to their counterparts. Thus, MOPT is effective in improving the performance of mutation-based fuzzers.

• *The fuzzers, which use symbolic execution or similar techniques, perform significantly better than others on LAVA-M.* MOPT again exhibits good compatibility and can be integrated with general mutation-based fuzzers. For instance, MOPT-Angora finds significantly more unique bugs than AFL-Angora, and MOPT improves the performance of AFL-QSYM in 3 cases. From Table 7, in addition to the compatibility, MOPT can find the unique bugs and paths which the symbolic execution fails to find.

In summary, MOPT-*based fuzzers perform much better*

Table 7: Evaluation on LAVA-M. The incremental number is the number of the discovered unlisted bugs.

Program	Listed bugs	Unlisted bugs	AFL Bugs	MOPT-AFL-ever Bugs	AFLFast Bugs	MOPT-AFLFast-ever Bugs	VUzzer Bugs	MOPT-VUzzer Bugs	AFL-Angora Bugs	MOPT-Angora Bugs	AFL-QSYM Bugs	MOPT-QSYM Bugs
base64	44	4	4	39	7	36	14	17	44(+2)	44(+3)	24	44(+4)
md5sum	57	4	2	23	1	18	38	41	57(+4)	57(+4)	57(+1)	57(+1)
uniq	28	1	5	27	7	15	22	24	26	28(+1)	1	18
who	2,136	381	1	5	2	6	15	23	1,622(+65)	2,069(+145)	312(+46)	774(+70)

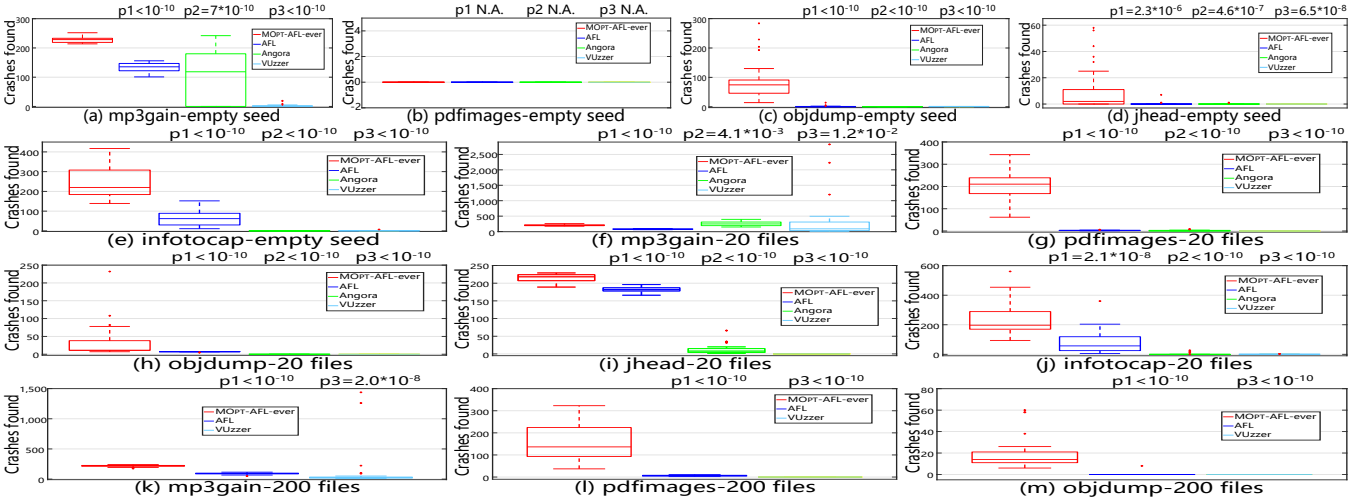

Figure 10: The boxplot generated by the number of unique crashes from 30 trials, which are found by AFL, MOPT-AFL-ever, Angora and VUzzer on five programs when fed with an empty seed, with 20 well-formed seed inputs and with 200 well-formed seed inputs. Y-axis: the number of unique crashes discovered in 24 hours.

compared with their counterparts. Furthermore, MOPT *is compatible with a wide range of techniques such as symbolic execution to improve the overall fuzzing performance.*

6 Further Analysis

6.1 Statistical Experiments with Different Seed Sets

Following the guidance of [38] and to make our evaluation more comprehensive, we conduct three extra groups of evaluations in this subsection. In the following, we detail the seed selection process, the evaluation methodology, and the analysis of the results.

Evaluation methodology and setup. To provide statistical evidences of our improvements, we measure the performance of MOPT-AFL-ever, AFL, Angora [10] and VUzzer [7] on five programs including `mp3gain`, `pdfimages`, `objdump`, `jhead` and `infotocap` (the detail of each program is shown in Table 2). Each program is tested by each fuzzer for 24 hours, on a virtual machine configured with one CPU core of 2.40Ghz E5-2640 V4, 4.5GB RAM and the OS of 64-bit Ubuntu 16.04 LTS. To eliminate the effect of randomness, we run each testing for 30 times.

To investigate the influence of the initial seed set on the performance of MOPT, we consider using various initial seed sets in our experiments such as an empty seed, or the seeds with different coverage, which are widely used in previous works [1, 6, 10, 21].

In the first group of experiments, each program is fed with an empty seed, which is a text file containing a letter 'a'. In the second and third groups of experiments, each program is fed with 20 and 200 well-formed seed inputs, respectively. In the third group, Angora is skipped since it reports errors to fuzz `pdfimages` when given 200 seed PDF files.

To obtain the seed inputs, we first download more than necessary (e.g., 1,700) input files with correct formats from the Internet. For example, we download mp3 files from the music download websites. Then, we split the input files (of format PDF and mp3) into a reasonable size if they are too large. Further, we utilize AFL-cmin [17] to evaluate each input file's coverage, and remove the inputs that have redundant coverage. In the remaining input files, we randomly select 20 (i.e., for the second group) or 200 (i.e., for the third group) seeds for the corresponding objective program.

Evaluation metrics. We measure the widely adopted metrics, i.e., *number of unique crashes* and *number of unique bugs*, to compare the performance of each fuzzer. To obtain the unique bugs, we recompile objective programs with the *AddressSanitizer* [37] instrumentation, and reevaluate the programs with the discovered crash inputs. If the top three

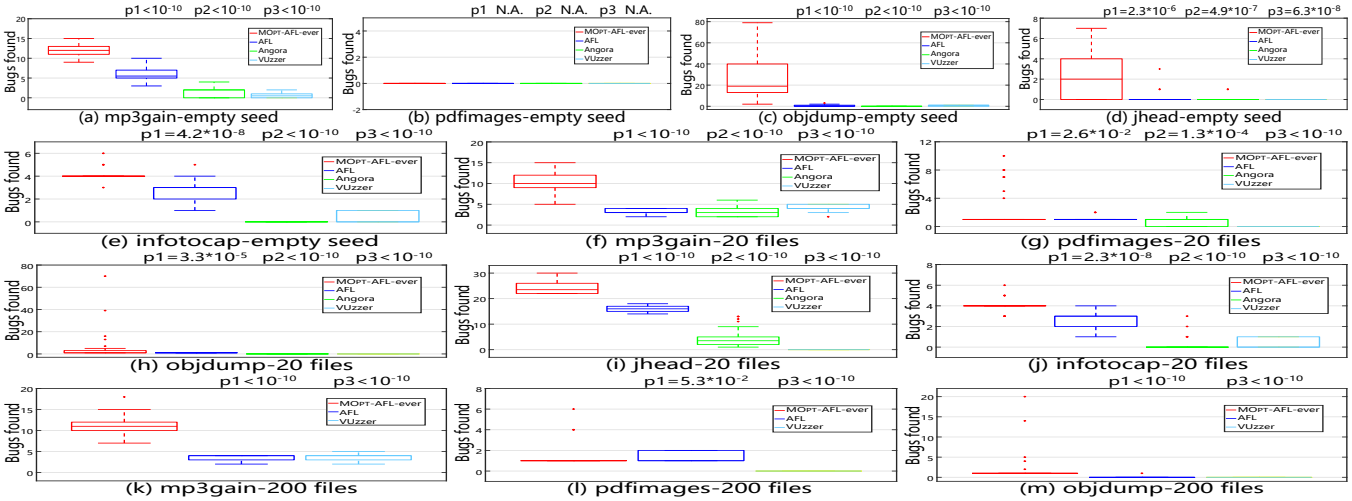

Figure 11: The boxplot generated by the number of unique bugs from 30 trials, which are found by AFL, MOPT-AFL-ever, Angora and VUzzer on five programs when fed with an empty seed, with 20 well-formed seed inputs and with 200 well-formed seed inputs. Y-axis: the number of unique bugs discovered in 24 hours.

source code locations of the stack trace provided by *AddressSanitizer* are unique, we consider the corresponding crash input triggers a unique bug of the objective program. This is a common way to find unique bugs in practice and has been used to calculate the stack hashing in [38].

Note that we do the statistical tests and use the p value [43] to measure the performance of the three fuzzers (suggested by [38]). In particular, $p1$ is the p value yielded from the difference between the performance of MOPT-AFL-ever and AFL, $p2$ is the p value yielded from the difference between the performance of MOPT-AFL-ever and Angora, and $p3$ is the p value generated from the difference between the performance of MOPT-AFL-ever and VUzzer.

Statistically, it is possible that the above p value test may introduce some false discoveries, i.e., a test may identify the minor performance difference between two fuzzers as significant difference. To decrease the false discovery rate, we further leverage the Benjamini-Hochberg (BH) procedure [44], which can screen out the tests with large difference from multiple tests and control the false discovery rate under a desired level of α. We define *BH proportion* as the ratio of the tests with large difference in all the tests. Specifically, to evaluate the performance difference between two fuzzers with the BH procedure, the process consists of the following three steps. First, for one test, we randomly select 25 results from 30 trials for each fuzzer. We repeat this random selection process for 1,000 times thus constituting 1,000 tests. Then, we utilize the BH procedure to screen out the tests with large difference and with the target of decreasing the false discovery rate to a level of no larger than $\alpha = 10^{-3}$. Finally, we calculate the BH proportion. If the corresponding BH proportion is large, i.e., most tests show large difference between two fuzzers with a low false discovery rate, we consider the performance of the two fuzzers as significantly different. The BH proportions between MOPT-AFL-ever and AFL, Angora, VUzzer are denoted by $BHP1$, $BHP2$ and $BHP3$, respectively.

Results and analysis. The number of unique crashes and unique bugs are shown in Fig. 10 and Fig. 11, respectively. From the results, we can learn the following facts.

• As shown in Fig. 10, among all the 13 evaluation settings, MOPT-AFL-ever discovers more unique crashes than the other fuzzers in 11 evaluations. In these 11 evaluations, $p1$, $p2$ and $p3$ are smaller than 10^{-5}, meaning that the distribution of the number of unique crashes discovered by MOPT-AFL-ever and the other fuzzers is widely different, which demonstrates a significant statistical evidence for MOPT's improvement. Therefore, according to the statistical results of 30 trials, MOPT-AFL-ever performs better than AFL, Angora and VUzzer in most cases.

• As for the number of unique bugs, MOPT-AFL-ever still performs significantly better than AFL, Angora and VUzzer in most cases. For instance, the minimum number of unique bugs discovered by MOPT-AFL-ever among the 30 runs is more than the maximum number of that discovered by other fuzzers when fuzzing `objdump` and `jead` with 20 files as the initial seed set. Further, we find that both Angora and VUzzer discover more unique crashes but fewer unique bugs than MOPT-AFL-ever when fuzzing `mp3gain` with the 20 files. This indicates that their deduplication strategies do not work well in this evaluation.

• When using an empty seed as the initial seed set to fuzz `pdfimages`, all the fuzzers cannot discover any unique crash. The reason is that PDF files have complex structures. The test cases mutated from an empty seed are hard to generate such complex structures, which leads to the poor fuzzing

Table 8: The p value and BH proportion in each evaluation.

	crashes						bugs					
	$p1$	$BHP1$	$p2$	$BHP2$	$p3$	$BHP3$	$p1$	$BHP1$	$p2$	$BHP2$	$p3$	$BHP3$
mp3gain (empty seed)	$<10^{-10}$	100.0%	$7*10^{-10}$	100.0%	$<10^{-10}$	100.0%	$<10^{-10}$	100.0%	$<10^{-10}$	100.0%	$<10^{-10}$	100.0%
pdfimages (empty seed)	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.	N.A.
objdump (empty seed)	$<10^{-10}$	100.0%	$<10^{-10}$	100.0%	$<10^{-10}$	100.0%	$<10^{-10}$	100.0%	$<10^{-10}$	100.0%	$<10^{-10}$	100.0%
jhead (empty seed)	$2.3*10^{-6}$	100.0%	$4.6*10^{-7}$	100.0%	$6.5*10^{-8}$	100.0%	$2.3*10^{-6}$	100.0%	$4.9*10^{-7}$	100.0%	$6.3*10^{-8}$	100.0%
infotocap (empty seed)	$<10^{-10}$	100.0%	$<10^{-10}$	100.0%	$<10^{-10}$	100.0%	$4.2*10^{-8}$	100.0%	$<10^{-10}$	100.0%	$<10^{-10}$	100.0%
mp3gain (20 files)	$<10^{-10}$	100.0%	$4.1*10^{-3}$	0.0%	$1.2*10^{-2}$	0.0%	$<10^{-10}$	100.0%	$<10^{-10}$	100.0%	$<10^{-10}$	100.0%
pdfimages (20 files)	$<10^{-10}$	100.0%	$<10^{-10}$	100.0%	$<10^{-10}$	100.0%	$2.6*10^{-2}$	0.0%	$1.3*10^{-4}$	74.1%	$<10^{-10}$	100.0%
objdump (20 files)	$<10^{-10}$	100.0%	$<10^{-10}$	100.0%	$<10^{-10}$	100.0%	$3.3*10^{-5}$	95.4%	$<10^{-10}$	100.0%	$<10^{-10}$	100.0%
jhead (20 files)	$<10^{-10}$	100.0%	$<10^{-10}$	100.0%	$<10^{-10}$	100.0%	$<10^{-10}$	100.0%	$<10^{-10}$	100.0%	$<10^{-10}$	100.0%
infotocap (20 files)	$2.1*10^{-8}$	100.0%	$<10^{-10}$	100.0%	$<10^{-10}$	100.0%	$2.3*10^{-8}$	100.0%	$<10^{-10}$	100.0%	$<10^{-10}$	100.0%
mp3gain (200 files)	$<10^{-10}$	100.0%			$2.0*10^{-8}$	100.0%	$<10^{-10}$	100.0%			$<10^{-10}$	100.0%
pdfimages (200 files)	$<10^{-10}$	100.0%			$<10^{-10}$	100.0%	$5.3*10^{-2}$	0.0%			$<10^{-10}$	100.0%
objdump (200 files)	$<10^{-10}$	100.0%			$<10^{-10}$	100.0%	$<10^{-10}$	100.0%			$<10^{-10}$	100.0%

Table 9: The results of AFL, MOPT-AFL-off, AFL-ever and MOPT-AFL-ever on 4 target programs.

Program	AFL		MOPT-AFL-off		AFL-ever		MOPT-AFL-ever	
	Unique crashes	Unique paths	Unique crashes	Unique paths	Unique crashes	Unique paths	Unique crashes	Unique paths
pdfimages	16	10,027	18	12,129	43	9,906	322	24,306
w3m	0	3,250	74	3,835	44	5,007	138	5,227
objdump	5	11,163	77	15,032	170	23,392	239	24,918
infotocap	86	3,179	97	4,112	436	6,808	687	7,109
total	107	27,619	266	35,108	693	45,113	1,386	61,560

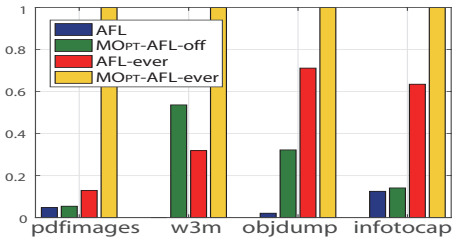

Figure 12: The ratio of the unique crashes discovered by 4 fuzzers, with MOPT-AFL-ever as the baseline.

performance. This reminds us the motivation of generation-based fuzzers and shows that: *although fuzzers like AFL may perform better with an empty seed, they cannot discover more crashes on the programs that require complex input formats when using an empty seed.*

• As shown in Table 8, the BH proportion is larger than 95.0% in most evaluations, demonstrating that our statistical results are reliable for most tests.

In summary, *statistically, the fuzzing performance of* MOPT-*AFL-ever is significantly better than AFL, Angora and VUzzer on the datasets in our evaluation, given the evidence of 30 trials per-experiment on five programs and with three different seed sets.*

6.2 Stepwise Analysis of MOPT Main Framework and Pacemaker Fuzzing Mode

To validate the effectiveness of MOPT main framework and the pacemaker fuzzing mode, we implement MOPT-AFL-off (that is based on MOPT-AFL-ever while disabling the pacemaker fuzzing mode) and AFL-ever (that is based on AFL and only implements the pacemaker fuzzing mode). We re-evaluate AFL, MOPT-AFL-off, AFL-ever and MOPT-AFL-ever on `pdfimages`, `w3m`, `objdump` and `infotocap` for 240 hours. The results are shown in Table 9.

MOPT *Main Framework (without Pacemaker Fuzzing Mode).* We can learn from Table 9 that MOPT-AFL-off discovers more crashes than AFL. For instance, on `w3m`, AFL cannot discover any crash in 240 hours, while MOPT-AFL-off discovers 74 unique crashes. Note that if without the pacemaker fuzzing mode, MOPT-AFL-off uses the havoc stage less frequently and iterates the selection distribution more slowly, which limits the performance of the MOPT main framework. Comparing MOPT-AFL-ever with AFL-ever, we can learn that MOPT-AFL-ever has a better capability to explore unique crashes than AFL-ever. As for the coverage, MOPT-AFL-off discovers more unique paths than AFL, and the same situation applies for MOPT-AFL-ever and AFL-ever. As a conclusion, both two comparison groups demonstrate that the MOPT scheme without the pacemaker fuzzing mode can also improve the performance of AFL on exploring unique crashes and paths, but a better performance can be achieved if integrating the pacemaker fuzzing mode.

Pacemaker Fuzzing Mode. For discovering unique crashes, AFL-ever discovers 165 more unique crashes than AFL on `objdump`. Additionally, MOPT-AFL-ever finds 1,120 more unique crashes than MOPT-AFL-off on the 4 programs in total. As for the coverage, AFL-ever is better than AFL on `w3m`, `objdump` and `infotocap`. MOPT-AFL-ever finds nearly twice as many unique paths as MOPT-AFL-off on all the programs except `w3m`. As a conclusion, the experiments demonstrate that the pacemaker fuzzing mode can help fuzzers find much more unique crashes and paths.

In summary, *both the* MOPT *main framework and pace-*

maker fuzzing mode can improve the fuzzing performance significantly, while the combination of both parts would result in an even better performance (corresponding to MOPT-*AFL-ever).* To further clarify this point, we use the number of unique crashes discovered by MOPT-AFL-ever as the baseline and observe the approximate fuzzing performance of each part. The results are shown in Fig. 12.

From the results, the improvement of the pacemaker fuzzing mode is relatively limited for fuzzing; however, without the pacemaker fuzzing mode, MOPT cannot converge fast to the proper selection probability distribution, which on the other hand limits the fuzzing performance either. Nevertheless, *the performance can be significantly improved if we combine AFL with the complete* MOPT *scheme.*

6.3 Iteration Analysis of Selection Probability

To demonstrate the effectiveness of MOPT in obtaining the proper selection probability for the mutation operators, we record the probability of `bitflip 1/1`, `arith 8/8` and `interest 16/8` obtained by the particles in one swarm when using MOPT-AFL-ever to fuzz `w3m` and `pdfimages`. The results are shown in Fig. 13, from which we have the following observations.

• Different mutation operators have different proper selection probabilities on each program. For instance, when fuzzing `pdfimages`, the proper selection probability of `arith 8/8` is around 0.04, while the proper probability of `bitflip 1/1` is around 0.065. Moreover, the proper selection probability of one mutation operator varies with the objective programs. For instance, the proper probability of `interest 16/8` is around 0.055 and 0.075 on `w3m` and `pdfimages`, respectively. The results are consistent with our motivation that it is desired to dynamically determine the selection probability of operators during the fuzzing process.

• G_{best} and L_{best} quickly converge to the proper values. For instance, it only takes an hour for G_{best} and L_{best} to converge to the proper values when fuzzing `w3m`. When the proper values of G_{best} and L_{best} are the same, x_{now} will converge to this value and oscillate around. Otherwise, x_{now} will oscillate between G_{best} and L_{best} to explore whether there is a better selection probability.

• We can learn from Fig. 13 that MOPT iterates slowly at first and iterates fast later when fuzzing `pdfimages`. The reasons are that (1) the deterministic stage is effective at finding unique crashes and paths in the early fuzzing time; (2) the fuzzer spends a long time on the deterministic stage of one test case when fuzzing `pdfimages`. When the efficiency of the deterministic stage decreases, i.e., it cannot discover any new crash or path for a long time, MOPT-AFL-ever enters the pacemaker fuzzing mode and will not use the deterministic stage again. Then the selection probability converges quickly and MOPT iterates fast. The results demonstrate that the design of the pacemaker fuzzing mode is reasonable and meaningful, which exploits the deterministic stage at first and avoids repeating its high computation when it is inefficient.

In summary, *the* MOPT *scheme generally converges fast and the design of the pacemaker fuzzing mode is effective.*

6.4 More Analysis in Appendix

Steadiness Analysis. To examine MOPT's steadiness, we repeat the experiments of MOPT-AFL for 4 times. The details are in Appendix A. In summary, MOPT-*AFL can maintain similar performance as in Section 5.3 in the repeated experiments, which demonstrates the stability of* MOPT.

Overhead Analysis. we analyze the execution efficiency of MOPT-AFL compared with AFL, whose details are in Appendix B. Based on the experimental results, *the computing overhead of* MOPT *is moderate and acceptable, and* MOPT-*AFL can even execute faster than AFL on some programs.*

Long Term Parallel Experiments. In order to verify the performance of MOPT-AFL in the long term parallel experiments, we run the three fuzzers of the same kind (AFL, MOPT-AFL-tmp and MOPT-AFL-ever) to fuzz `pdfimages` in parallel, whose details are in Appendix C. The results show that MOPT-AFL has outstanding performance in the long term parallel experiments.

7 Limitation and Discussion

In order to further analyze the compatibility of MOPT, we are eager to combine it with state-of-the-art fuzzers such as CollAFL [5] and Steelix [11] after they open-source their system code. By leveraging MOPT as an optimal strategy for selecting mutation operators, we believe the performance of these systems can be further enhanced.

In our evaluation, we consider 13 real world programs and several seed selection strategies, which are still a limited number of scenarios. In our evaluation, overall, MOPT-AFL discovers 31 vulnerabilities on `tiff2bw` and `sam2p` and 66 unreported CVEs on the other 11 programs. Furthermore, both MOPT-Angora and MOPT-QSYM perform better than previous methods on the benchmark dataset LAVA-M. Therefore, the proposed MOPT is promising to explore vulnerabilities for real world programs. Nevertheless, the performance advantage exhibited in our evaluation may not be applicable to all the possible programs and seeds. Our evaluation can be enhanced by further conducting more in-depth evaluation in large-scale. To make our evaluation more comprehensive, we are planning to perform a large-scale evaluation of MOPT using more real world programs and benchmarks in the future.

As a future work, it is interesting to investigate better mutation operators to further enhance the effectiveness of MOPT. Constructing a more comprehensive and represen-

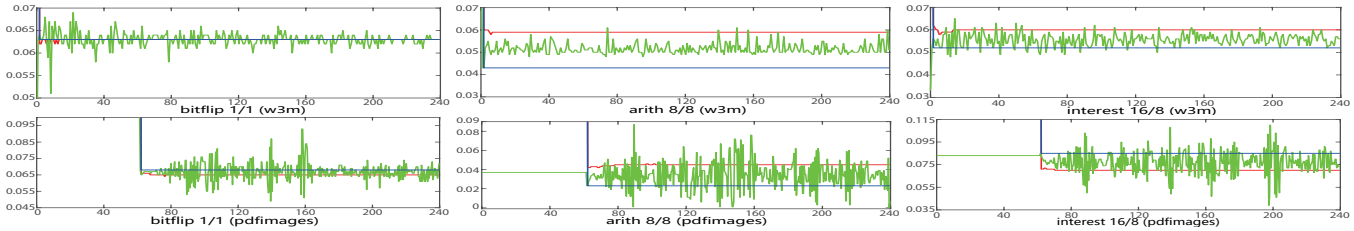

Figure 13: The probability when using MOPT-AFL-ever to fuzz `w3m` and `pdfimages`. X-axis: time (over 240 hours). Y-axis: the selection probability of the corresponding mutation operator. Green line: x_{now}. Red line: G_{best}. Blue line: L_{best}.

tative benchmark dataset to systematically evaluate the performance of fuzzers is another interesting future work.

8 Related Work

In this section, we summarize the existing fuzzing mechanisms and the related seed selection strategies.

Mutation-based fuzzing. AFL is one of the most well-recognized fuzzers because of its high-efficiency and ease of use [17]. Multiple efficient fuzzers were developed based on AFL [5, 6]. To improve the fuzzing performance, some combined the mutation-based fuzzing with other bug detection technologies [14, 15, 16, 45]. Another method to improve mutation-based fuzzers is coverage-based fuzzing [7, 11, 12]. Yun et al. presented a fast concolic execution engine named *QSYM* to help fuzzers explore more bugs and paths [42]. By solving the path constraints without symbolic execution, *Angora* presented by Chen et al. can significantly increase the branch coverage of programs [10].

MOPT presented in our paper is a scheme of improving the test case mutation process and generating high-quality mutated test cases. Taking the advantage of its compatibility, it can be combined with most of the aforementioned fuzzers.

Although in this paper we focus on using MOPT to improve mutation-based fuzzers, it can also be implemented in other kinds of fuzzers, such as generation-based fuzzers and kernel fuzzers, if they have the issues to select proper operators to generate test cases. MOPT can also be combined with most existing seed selection strategies since they can provide better initial seed sets for fuzzers. We briefly introduce the state-of-the-art related works in these area as follows.

Generation-based fuzzing. Generation-based fuzzers focus on the programs that require the test cases with specific input formats [46, 47, 48]. Recently, Wang et al. presented a novel data-driven seed generation approach named *Skyfire* to generate interesting test cases for XML and XSL [1]. Godefroid et al. presented a RNN-based machine learning technique to automatically generate a grammar for the test cases with complex input formats [49].

Other fuzzing strategies. Several works presented effective kernel fuzzers [50, 51]. Xu et al. [13] implemented three new operating primitives to benefit large-scale fuzzing and cloud-based fuzzing services. You et al. presented *SemFuzz* to learn from vulnerability-related texts and automatically generate Proof-of-Concept (PoC) exploits [52]. Petsios et al. proposed *SlowFuzz* to trigger algorithmic complexity vulnerabilities [35]. Klees et al. performed extensive experiments and proposed several guidelines to improve the experimental evaluations for fuzzing [38]. Some works proposed state-of-the-art directed greybox fuzzers to rapidly reach the target program locations [21, 53]. Recently, several works [8], [9] employ the reinforcement learning algorithms as the mutation schedulers and propose their fuzzing frameworks, respectively. However, the performance improvement of these methods is limited based on their experimental results.

Seed selection strategies. Several works focused on how to select a better seed set [2, 3, 4]. Nichols et al. showed that using the generated files of GAN to reinitialize AFL can find more unique paths of `ethkey` [54]. Lv et al. presented *SmartSeed* to leverage machine learning algorithms to generate high-quality seed files for different input formats [55].

9 Conclusion

We first studied the issues of existing mutation-based fuzzers which employ the uniform distribution for selecting mutation operators. To overcome these issues, we presented a mutation scheduling scheme, named MOPT, based on Particle Swarm Optimization (PSO). By using MOPT to search the optimal selection distribution for mutation operators and leveraging the pacemaker fuzzing mode to further accelerate the convergence speed of searching, MOPT can efficiently and effectively determine the proper distribution for selecting mutation operators. Our evaluation on 13 real-world applications demonstrated that MOPT-based fuzzers can significantly outperform the state-of-the-art fuzzers such as AFL, AFLFast and VUzzer in most cases. We also conducted systematic analysis to demonstrate the rationality, compatibility, low cost characteristic and steadiness of MOPT. Our fuzzers found 81 security CVEs on 11 real world programs, of which 66 are the newly reported CVEs. Overall, MOPT can serve as a key enabler for mutation-based fuzzers in discovering software vulnerabilities, crashes and program paths.

Acknowledgments

We sincerely appreciate the shepherding from Adam Doupé. We would also like to thank the anonymous reviewers for their valuable comments and input to improve our paper.

References

[1] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *S&P*, 2017.

[2] A. D. Householder and J. M. Foote, "Probability-based parameter selection for black-box fuzz testing," CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, Tech. Rep., 2012.

[3] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *CCS*, 2013.

[4] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing." in *USENIX*, 2014.

[5] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *S&P*, 2018.

[6] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *CCS*, 2016.

[7] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *NDSS*, 2017.

[8] K. Böttinger, P. Godefroid, and R. Singh, "Deep reinforcement fuzzing," *arXiv preprint arXiv:1801.04589*, 2018.

[9] W. Drozd and M. D. Wagner, "Fuzzergym: A competitive framework for fuzzing and learning," *arXiv preprint arXiv:1807.07490*, 2018.

[10] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *S&P*, 2018.

[11] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," in *FSE*, 2017.

[12] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in *S&P*, 2018.

[13] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing new operating primitives to improve fuzzing performance," in *CCS*, 2017.

[14] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: a guided fuzzer to find buffer boundary violations." in *USENIX*, 2013.

[15] S. K. Cha, M. Woo, and D. Brumley, "Program-adaptive mutational fuzzing," in *S&P*, 2015.

[16] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, 2016.

[17] "American fuzzy lop," http://lcamtuf.coredump.cx/afl/.

[18] K. Serebryany, "Continuous fuzzing with libfuzzer and addresssanitizer," in *SecDev*, 2016.

[19] R. Swiecki, "Honggfuzz," *Available online a t: http://code. google. com/p/honggfuzz*, 2016.

[20] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *MHS*, 1995.

[21] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *CCS*, 2017.

[22] "Bento4," https://github.com/axiomatic-systems/Bento4.

[23] "exiv2," https://github.com/Exiv2/exiv2.

[24] "mp3gain," https://sourceforge.net/projects/mp3gain/.

[25] "libtiff," https://gitlab.com/libtiff/libtiff.

[26] "xpdf," http://www.xpdfreader.com/.

[27] "sam2p," https://github.com/pts/sam2p.

[28] "libav," https://github.com/libav/libav.

[29] "w3m," https://sourceforge.net/projects/w3m/.

[30] "binutils," http://www.gnu.org/software/binutils/.

[31] "jhead," http://www.sentex.net/~mwandel/jhead/.

[32] "mpg321," https://sourceforge.net/projects/mpg321/.

[33] "ncurses," http://invisible-island.net/ncurses/.

[34] "podofo," http://podofo.sourceforge.net/.

[35] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *CCS*, 2017.

[36] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "Nezha: Efficient domain-independent differential testing," in *S&P*, 2017.

[37] "Addresssanitizer," http://clang.llvm.org/docs/AddressSanitizer.html.

[38] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *CCS*, 2018.

[39] "Common vulnerability scoring system (cvss)," https://www.first.org/cvss.

[40] "Cve details," https://www.cvedetails.com/.

[41] B. Dolangavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *S&P*, 2016.

[42] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "Qsym: A practical concolic execution engine tailored for hybrid fuzzing," in *USENIX*, 2018.

[43] "p value," https://en.wikipedia.org/wiki/P-value.

[44] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: a practical and powerful approach to multiple testing," *J R STAT SOC B*, 1995.

[45] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *S&P*, 2010.

[46] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *ACM Sigplan Notices*, 2008.

[47] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments." in *USENIX*, 2012.

[48] K. Dewey, J. Roesch, and B. Hardekopf, "Language fuzzing using constraint logic programming," in *ASE*, 2014.

[49] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *ASE*, 2017.

[50] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "Difuze: interface aware fuzzing for kernel drivers," in *CCS*, 2017.

[51] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *CCS*, 2017.

[52] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang, "Semfuzz: Semantics-based automatic generation of proof-of-concept exploits," in *CCS*, 2017.

[53] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *CCS*, 2018.

[54] N. Nichols, M. Raugas, R. Jasper, and N. Hilliard, "Faster fuzzing: Reinitialization with deep neural models," *arXiv preprint arXiv:1711.02807*, 2017.

[55] C. Lv, S. Ji, Y. Li, J. Zhou, J. Chen, P. Zhou, and J. Chen, "Smartseed: Smart seed generation for efficient fuzzing," *arXiv preprint arXiv:1807.02606*, 2018.

A Repetitive Experiments in Section 5.3

To examine the fuzzing steadiness, we test AFL, MOPT-AFL-tmp and MOPT-AFL-ever on `pdfimages`, `w3m`, `objdump` and `infotocap` for 4 times, each of which lasts for 240 hours and has the same settings as in Section 5.3. The results are shown in Table 10, from which we can learn the following conclusions.

• For most tests of a program, the number of unique crashes discovered by each fuzzer is on the same order of magnitude. For instance, when fuzzing `pdfimages`, AFL discovers around 16 unique crashes in the four experiments, MOPT-AFL-tmp discovers around 340 crashes, and MOPT-AFL-ever finds around 346 crashes for three times. Therefore, although the fuzzing process is accompanied by randomness and the number of discovered crashes may be floating, each fuzzer tends to be stable in a relatively long test time and finds crashes with the same order of magnitude.

• As for the coverage, each fuzzer discovers the similar number of unique paths on one objective program in the repeated experiments. For instance, when fuzzing `w3m`, AFL finds nearly 3,300 paths, MOPT-AFL-tmp finds around 5,400 paths, and MOPT-AFL-ever finds around 5,300 paths in the four experiments. Again, the number of discovered paths is also pretty stable in the experiments.

According to the results in this subsection, both MOPT-AFL fuzzers and AFL exhibit similar performance as in Section 5.3 in the repeated experiment traces, which demonstrates the good stability of MOPT.

B Overhead Analysis

In this subsection, we analyze the execution efficiency of the MOPT-AFL fuzzers.

In order to compare the execution efficiency of each fuzzer in Section 5.3, we collect the total execution times of each fuzzer, which are the times a fuzzer uses the generated test cases to test an objective program, within 240 hours. The results are shown in Table 11, from which we learn the following conclusions.

Although MOPT-AFL fuzzers take partial computing power to improve the mutation scheduler, the execution efficiency of MOPT-AFL-tmp and MOPT-AFL-ever is still comparable with AFL on most programs. In many cases,

Table 10: The steadiness performance of AFL, MOPT-AFL-tmp and MOPT-AFL-ever.

Fuzzers		pdfimages		w3m		objdump		infotocap	
		Unique crashes	Unique paths	Unique crashes	Unique paths	Unique crashes	Unique paths	Unique crashes	Unique paths
AFL	1	23	12,915	0	3,243	0	11,565	92	3,710
	2	16	10,027	0	3,250	5	11,163	86	3,179
	3	10	10,356	25	3,270	2	10,709	18	3,328
	4	13	9,850	0	3,673	2	12,410	47	3,486
MOPT-AFL-tmp	1	357	22,661	506	5,313	470	19,309	340	6,157
	2	318	22,193	311	5,381	235	16,055	590	6,802
	3	316	20,382	223	5,420	333	21,511	711	6,984
	4	379	21,815	400	5,429	365	17,195	713	7,153
MOPT-AFL-ever	1	471	26,669	182	5,326	287	22,648	692	7,048
	2	322	24,306	138	5,227	239	24,918	687	7,109
	3	346	23,759	270	5,512	310	22,588	578	6,761
	4	379	24,100	300	5,187	601	24,541	651	7,224

Table 11: The total execution times and executions per second of AFL, MOPT-AFL-tmp and MOPT-AFL-ever.

Program	AFL		MOPT-AFL-tmp			MOPT-AFL-ever		
	Total execution times	Executions per second	Total execution times	Executions per second	Increase	Total execution times	Executions per second	Increase
mp42aac	**127.1M**	**147.12**	126.8M	146.71	-0.28%	124.6M	144.26	-1.94%
exiv2	35.1M	40.58	27.6M	31.89	-21.41%	**46.5M**	**53.83**	+32.65%
mp3gain	**182.2M**	**210.90**	117.2M	135.60	-35.70%	121.4M	140.53	-33.38%
tiff2bw	**906.7M**	**1,049.43**	613.2M	709.74	-32.37%	623.4M	721.55	-31.24%
pdfimages	91.7M	106.17	88.8M	102.80	-3.17%	**108.5M**	**125.59**	+18.29%
sam2p	42.6M	49.34	**52.3M**	**60.58**	+22.78%	28.9M	33.47	-32.16%
avconv	**48.6M**	**56.27**	43.3M	50.08	-11.00%	42.0M	48.61	-13.61%
w3m	104.4M	120.78	123.2M	142.64	+18.10%	**204.6M**	**236.75**	+96.02%
objdump	383.7M	444.13	436.7M	505.42	+13.80%	**843.8M**	**976.58**	+119.89%
jhead	418.5M	484.41	1,372.6M	1,588.63	+227.95%	**1,476.1M**	**1,708.40**	+252.68%
mpg321	119.7M	138.52	158.1M	182.94	+32.07%	**165.2M**	**191.17**	+38.01%
infotocap	**218.1M**	**252.41**	157.1M	181.88	-27.94%	199.9M	231.36	-8.34%
podofopdfinfo	379.6M	439.37	**411.3M**	**476.05**	+8.35%	340.2M	393.80	-10.37%
average	254.8M	294.95	310.7M	359.59	+15.93%	**360.43M**	**417.16**	+35.54%

e.g., when fuzzing `mp3gain`, `tiff2bw` and `mp42aac`, although the MOPT-AFL fuzzers test the objective programs for fewer times, they find much more crashes and paths than AFL.

Interestingly, MOPT-AFL can execute the tests faster than AFL on several programs. For instance, MOPT-AFL-tmp executes 22.78% more times than AFL on `sam2p`. MOPT-AFL-ever executes 252.68% more times than AFL on `jhead`. Moreover, the MOPT-AFL yields a better average execution efficiency on the 13 programs. We analyze the reasons as follows. The execution speed of each test case is different, and thus the test cases with slow execution speed will take more time consumption. When the fuzzing queue of AFL contains slow test cases, it will generate a number of test cases mutated from the slow test cases in the deterministic stage, which may also be executed slowly with a high probability and decrease the execution efficiency of AFL. As for MOPT-AFL fuzzers, they will generate much fewer mutated cases from the slow test cases since they tend to disable the deterministic stage when it is not efficient. Therefore, MOPT-AFL fuzzers will spend much less time on the slow test cases, followed by yielding a high execution efficiency.

In summary, the computing overhead of MOPT is moderate and acceptable, and the fuzzers with MOPT can even execute faster on some programs.

Table 12: The performance of AFL, MOPT-AFL-tmp and MOPT-AFL-ever in the long term parallel experiments when fuzzing `pdfimages`.

		Fuzzer1	Fuzzer2	Fuzzer3	Total
AFL	Unique crashes	11	871	896	1,778
	Unique paths	24,763	29,329	29,329	83,421
MOPT-AFL-tmp	Unique crashes	834	1,031	1,042	2,907
	Unique paths	30,098	31,600	31,520	93,218
MOPT-AFL-ever	Unique crashes	723	974	1,005	2,702
	Unique paths	28,047	30,910	30,966	89,923

C Long Term Parallel Experiments

In this subsection, we run the long term parallel experiments in order to verify the performance of MOPT-AFL in parallel for a long time. In each experiment, AFL, MOPT-AFL-tmp and MOPT-AFL-ever, are employed to fuzz `pdfimages` in parallel. Each experiment has three instances denoted by Fuzzer1, Fuzzer2 and Fuzzer3, with 20 carefully selected PDF files filtered from AFL-cmin [17] as the initial seed set. According to the parallel design of AFL and MOPT-AFL, the Fuzzer1 of AFL, MOPT-AFL-tmp and MOPT-AFL-ever will still perform the deterministic stage, while their Fuzzer2 and Fuzzer3 will disable it in the parallel experiments. Each experiment runs on a virtual machine configured with four CPU cores of 2.40Ghz E5-2640 V4, 4.5

GB RAM and the OS of 64-bit Ubuntu 16.04 LTS. The total CPU time of each experiment exceeds 70 days till the writing of this report and AFL, MOPT-AFL-tmp and MOPT-AFL-ever discover 1,778, 2,907 and 2,702 unique crashes, respectively.

The results are shown in Table 12, from which we can see that AFL's performance of discovering unique crashes is obviously inferior to MOPT-AFL's. Fuzzer1 of AFL enables the deterministic stage all the time and only discovers 11 unique crashes in more than 23 days, demonstrating the inefficiency of the deterministic stage. What's more, the performance of Fuzzer1 of MOPT-AFL-tmp is much better than that of MOPT-AFL-ever and AFL. We conjecture the reasons as follows. Since PDF files require the strict file format, there are many unique execution paths in `pdfimages` that contain strict magic byte checks. The operators, e.g., `bitflip 1/1`, in the deterministic stage are better at generating the correct magic bytes since fuzzers will flip every bit in the current test case to generate new test cases. In the later time, MOPT-AFL-tmp will enable the deterministic stage again while MOPT-AFL-ever will not. Thus MOPT-AFL-tmp is better at discovering unique paths containing magic byte checks than MOPT-AFL-ever. As for AFL, since it will go through the deterministic stage for all the test cases, it spends most time on this stage and discovers few unique crashes and paths. While MOPT-AFL-tmp will disable the deterministic stage when it cannot discover any interesting test case for a long time, after some time, it will re-enable the deterministic stage again and will perform the deterministic stage with the widely different test cases in the fuzzing queue. Therefore, MOPT-AFL-tmp can keep efficient fuzzing performance and can perform the deterministic stage on widely different test cases.

We can also observe from Table 12 that AFL's Fuzzer2 and Fuzzer3 find much more unique crashes than its Fuzzer1 without of the deterministic stage. The Fuzzer1 of MOPT-AFL-tmp and MOPT-AFL-ever finds much more crashes than AFL's Fuzzer1 and suppresses the performance of Fuzzer2 and Fuzzer3 in some way. Meantime, the Fuzzer2 and Fuzzer3 of both MOPT-AFL-tmp and MOPT-AFL-ever perform better than those of AFL. All these results again demonstrate the improvement of the customized PSO algorithm.

As a conclusion, both MOPT-AFL-tmp and MOPT-AFL-ever perform significantly better than AFL in the long term parallel experiments. The pacemaker fuzzing mode used in MOPT-AFL-tmp is better at passing the magic byte checks in the programs that require complex input formats.