MobFuzz: Adaptive Multi-objective Optimization in Gray-box Fuzzing

Gen Zhang, Pengfei Wang✉, Tai Yue, Xiangdong Kong, Shan Huang, Xu Zhou, Kai Lu✉
College of Computer, National University of Defense Technology
{zhanggen, pfwang, yuetai17, kongxiangdong, huangshan12, zhouxu, kailu}@nudt.edu.cn

***Abstract*—Coverage-guided gray-box fuzzing (CGF) is an efficient software testing technique. There are usually multiple objectives to optimize in CGF. However, existing CGF methods cannot successfully find the optimal values for multiple objectives simultaneously. In this paper, we propose a gray-box fuzzer for multi-objective optimization (MOO) called MobFuzz. We model the multi-objective optimization process as a multi-player multi-armed bandit (MPMAB). First, it adaptively selects the objective combination that contains the most appropriate objectives for the current situation. Second, our model deals with the power schedule, which adaptively allocates energy to the seeds under the chosen objective combination. In MobFuzz, we propose an evolutionary algorithm called NIC to optimize our chosen objectives simultaneously without incurring additional performance overhead. To prove the effectiveness of MobFuzz, we conduct experiments on 12 real-world programs and the MAGMA data set. Experiment results show that multi-objective optimization in MobFuzz outperforms single-objective fuzzing in the baseline fuzzers. In contrast to them, MobFuzz can select the optimal objective combination and increase the values of multiple objectives up to 107%, with at most a 55% reduction in the energy consumption. Moreover, MobFuzz has up to 6% more program coverage and finds 3x more unique bugs than the baseline fuzzers. The NIC algorithm has at least a 2x improvement with a performance overhead of approximately 3%.**

I. Introduction

Fuzz testing, or fuzzing, is one of the most successful search-based software testing approaches. Coverage-guided gray-box fuzzing (CGF), as an important variant of fuzzing, has recently received wide attention from researchers[1]. Essentially, CGF is an **optimization** problem [2], [3]. The key to an optimization approach is to search the input space to find optimal solutions and optimize the **objective**s. Optimizing the objectives means searching for the inputs that maximize or minimize the values of objectives [4]. In CGF, the most significant objective is code coverage, and the goal of CGF is to maximize coverage.

Single-objective optimization searches for the optimal solutions for only one objective. However, in real-world situations, more than one objective is required to be optimized simultaneously to solve difficult problems[4], [5], such as detecting different kinds of bugs and improving fuzzing efficiency. Specifically, in different stages of the fuzzing process, these objectives should be adaptively selected and prioritized according to the testing scenario. For example, when testing code fragments of memory allocation, seeds regarding the objective of memory consumption should be prioritized; to break the embedded branch conditions, seeds with more satisfied comparison bytes should be an important objective. Thus, multi-objective optimization (MOO) is proposed to effectively study the balanced solutions with optimal trade-offs among multiple objectives [6], [3], [4].

Though coverage-guided fuzzers also consider objectives other than coverage in the searching process, existing gray-box fuzzers cannot really support multi-objective optimization. AFL [7], for example, also searches for inputs with two other objectives, the execution time and input size. Favorable inputs (i.e., seeds) that have a smaller product of these two objectives (`speed` $*$ `size`) are selected. Theoretically, in the search process, considering one solution at a time, e.g., the product of the objectives, may result in getting stuck in a local optimum and being unable to produce the global optimal solution [3]. Some tools cannot coordinate multiple objectives simultaneously. When adding new objectives, old ones are discarded. For example, MemLock [8] targets memory consumption bugs by choosing seeds with more memory consumption. It optimizes the objectives of both coverage and memory consumption. However, as a tool based on AFL, MemLock entirely removes the speed objective of AFL. This ignorance of multiple objectives clearly affects the execution speed of fuzzing according to our experiments.

Therefore, in reality, to properly optimize multiple objectives simultaneously in CGF, we have to overcome the following challenges: 1) **Conflict effects among different objectives.** During the long campaign of fuzzing, optimizing one objective may have a negative effect on another objective. For example, according to our experiments, pushing the number of satisfied comparison bytes to a large value to pass branch conditions will slow down the whole fuzzing process. This conflicting internal relationship among objectives requires us to properly coordinate different objectives in different stages to search for a global optimum solution. Thus, we conclude *the adaptive selection of objective combination* as the first research point in this paper.

2) **Power schedule suitable for a multi-objective situation.** The power schedule of CGF is used to control the number of mutations and executions (i.e., amount of energy) on seeds [9], which directs the fuzzing process. Previous work on power schedule, such as AFLFast [1] and EcoFuzz [10], aims to allocate an appropriate amount of energy based on the path discovery ability of seeds to save energy. However, under

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.24314
www.ndss-symposium.org

the multi-objective situation, the power schedule is required to combine with objective combination selection to control the energy allocation. Thus, we identify *power schedule combined with objective combination selection* as the second research point in this paper.

3) **Reducing performance overhead introduced by multi-objective fuzzing.** Efficiency is an important metric in fuzzing. When taking multi-objectives into consideration, the objective combination selection, as well as the optimization on power schedule and mutation strategy all introduce additional overhead. For example, Cerebro[11] uses the idea of the Pareto frontier (i.e., the set of seeds with the optimal objective values) and non-dominated sorting [12] to search for the optimum solutions in an evolutionary process. However, this process of Cerebro is executed only once in a fuzzing cycle. It keeps the fuzzer waiting for the final result and wastes precious CPU time. Additionally, a single run cannot produce the global optimal solution. Calculating the Pareto frontier and revealing the convergence usually require more than 100 iterations through the evolutionary process[12]. Directly adopting this evolutionary procedure in fuzzing to find the optimal solution will bring significant performance overhead. Therefore, the third research point is to *find the optimal results of the selected objectives without introducing additional performance overhead*.

To overcome the above challenges of gray-box fuzzing in MOO, in this paper, we propose MobFuzz. To deal with the adaptive selection of objective combinations, we model the process of CGF under multi-objectives as a multi-player multi-armed bandit (MPMAB) problem. The goal of the classic MAB model is to maximize the reward in finite trials by choosing the appropriate arms[13], [10]. We model the objective combinations as different players with their own goals to deal with the problem of combination selection. The best objective combination that has the maximum reward for the current fuzzing state is selected. To deal with the power schedule suitable for a multi-objective situation, we model the seeds as bandit's arms and classify the fuzzing states into exploration and exploitation. MobFuzz controls the number of mutations and executions on a seed through the adaptive power schedule. Using this model, MobFuzz allocates the appropriate amount of energy for seeds under the chosen combination to reach the optimal results and avoid a waste of energy. To address the third challenge, we propose an evolutionary algorithm called non-dominated sorting genetic algorithm in CGF (**NIC**). It is designed based on the Pareto frontier and non-dominated sorting to search for optimal solutions in an evolutionary process. A new mutation strategy is designed by casting the objective combination to a corresponding mutation operator combination. The mutators that are more likely to increase the objective values are selected with a greater chance. In addition, we propose several methods, such as a shared seed pool, to reduce the performance overhead.

To prove the effectiveness of MobFuzz, we conduct a series of experiments on real-world target programs and the MAGMA data set. Experiment results show that multi-objective optimization in MobFuzz can outperform single-objective optimization in the baselines. Compared with the state-of-the-art fuzzers, such as MemLock and FuzzFactory, MobFuzz can optimize all the objectives simultaneously to reach the optimal values. Specifically, MobFuzz exceeds its competitors up to 107% in the values of objectives. In addition, the results show the effectiveness of our MPMAB model and NIC algorithm. We reduce at most 55% energy consumption, and NIC has at least a 2x performance improvement compared with the baseline fuzzers with only 3.3% performance overhead. Additionally, MobFuzz has at most 6% more program coverage and finds 3x more bugs than the competitors. In conclusion, we make the following contributions in this paper:

- We target the weakness of CGF in multi-objective optimization. We model the MOO in gray-box fuzzing as a multi-player MAB problem and adaptively select the objective combinations and allocate energy to seeds by the model.
- We propose the NIC algorithm in MobFuzz to solve the problems of existing fuzzers in finding the optimal results. NIC is integrated into the fuzzing loop and searches for the optimal objective values without introducing too much overhead.
- We implement MobFuzz and evaluate it with real-world programs and the MAGMA data set. The results demonstrate the effectiveness of multi-objective optimization in CGF.

II. Background

A. CGF and Objectives

As one of the most popular software testing techniques, fuzzing has recently developed rapidly, especially in the field of coverage-guided gray-box fuzzing[14], [15], [16], [17], [18]. Compared with the plain black-box fuzzing and complicated white-box fuzzing, the key of CGF is to maximize code coverage through lightweight instrumentation[9]. As a representative of CGF, AFL exposed many security-critical vulnerabilities with these features[19].

Basic infrastructure. CGF starts with selecting a seed from the seed pool according to its goal and then allocates energy to the selected seeds via the power schedule. The energy controls the number of mutations and executions to this seed. Next, the seed is mutated to generate test cases. When executing these test cases, CGF monitors whether they achieve new code coverage. Test cases that achieve new coverage will be saved as seeds to the seed pool. Later, CGF goes back to seed selection and starts the next round of fuzzing.

Objectives in CGF. In addition to code coverage, gray-box fuzzers usually need to maximize multiple objectives when maintaining the seed pool. For instance, AFL searches for seeds with a faster execution speed and smaller size. The product of them, i.e., `speed` $*$ `size`, is used to optimize the objectives. Seeds with a larger product result will be marked as **favored**, and they will be selected with a greater chance than the non-favored seeds.

However, existing coverage-guided fuzzers cannot support multi-objective optimization. Some fuzzers based on AFL disable the original speed objective in AFL when adding a new objective, such as MemLock [8] and FuzzFactory [20]; other solutions use simple algorithms to coordinate multiple objectives, which will get stuck at a local optimum [7], or

introduce unacceptable overhead [11]. Therefore, a solution that can optimize multiple objectives simultaneously without causing performance reduction is needed.

B. MAB Problem

Exploration vs. Exploitation. The trade-off between exploration and exploitation is an important concept in game theory[13]. Inspired by a player's choice to maximize the reward when playing a slot machine, the multi-armed bandit model is proposed to solve this problem[21]. According to the definition, a classic MAB model contains `N` parallel arms, and only one arm is selected each time. The expectation of reward for arm `i` ($\texttt{i} \in \{1, 2, ..., \texttt{N}\}$) is defined as $\texttt{R}_\texttt{i}$. The key to the MAB problem is to maximize the total reward within finite arm selections. However, before trying a certain arm, its reward is unknown. The process of **exploration** performs trials on an arm to acquire a more accurate calculation of its reward. When the rewards of all the arms are known, choosing the arm with the maximum reward is the process of **exploitation**. In conclusion, choosing the best arm (exploitation) will maximize the current total reward. In the long term, making trials on reward-unknown arms (exploration) will help reach a larger total reward [22]. It is our goal to weigh the strengths and weaknesses between exploration and exploitation in the MAB problem.

The inappropriateness of using MAB in CGF. Previous work in fuzzing, such as EcoFuzz [10], models seeds as arms in MAB and solves specific problems through this model, e.g., energy allocation. If we want to make improvements on MOO in CGF, the classic MAB model is inappropriate. First, as discussed above, there are two choices we need to make: objective combination selection and energy allocation. It is natural to consider the seeds as arms in the MAB model. However, a single-player MAB model is no longer applicable since different objective combinations stand for different players with their own goals. In other words, we need a multi-player MAB to model this process. Second, the reward of the classic MAB model is time-invariant, and the number of arms is constant[10]. However, as the fuzzing campaign continues, the rewards of the objective combinations and seeds will change accordingly. Additionally, the number of seeds is not constant during fuzzing. The above drawbacks of the classic MAB model drive us to propose a variant model of MAB that is suitable for multi-objectives in fuzzing.

III. Adaptive Multi-objective Optimization

A. Overview

As depicted in Figure 1, MobFuzz is designed by adding two new modules to the classic fuzzing process: the **MPMAB** model and the **NIC** algorithm. The MPMAB model adaptively determines the objective combination and energy allocation. The NIC algorithm produces the optimal objective values through an evolutionary process without introducing additional performance overhead.

The basic procedure of MobFuzz contains the following steps. First, a seed is selected from the seed pool. Next, the MPMAB model determines the best objective combination under the current fuzzing status. Based on the chosen objective combination, MPMAB allocates different amounts of energy

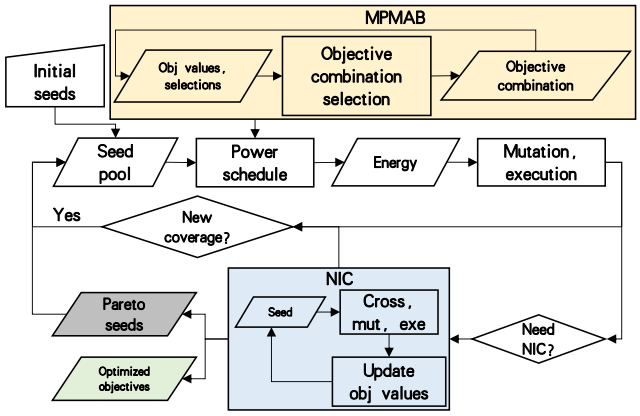

Fig. 1. The main fuzzing loop of MobFuzz. The sub-processes in different colors are our key approaches in MobFuzz.

to seeds in exploration and exploitation states. Then, MobFuzz performs mutations and executions based on the allocated energy. At the same time, it monitors the objective values in the chosen combination. If a starting condition is met, the NIC algorithm is invoked. NIC is an evolutionary process. It updates the objective values to gradually approach the optimal solutions. Finally, the Pareto seeds with the optimal values are saved into the seed pool, and the next round of fuzzing begins.

B. Multi-player Multi-armed Bandit Model

Our MPMAB model deals with two problems, including adaptively selecting objective combinations and allocating energy according to the chosen objective combination.

TABLE I. The names and definitions of variables in the MPMAB model

Name	Definition
t	the ID of the current fuzzing round
O_i	the ith objective
C_l	the lth combination
$v^t_{O_i}$, $v^t_{C_l}$	the average value of objective O_i or the objectives in C_l in round t
R	the reward
s_j	the jth seed
$v_{O_i}(s_j)$	the value of objective O_i after executing s_j

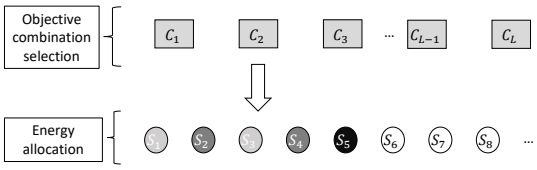

Fig. 2. Demonstration of the MPMAB model. The rectangles indicate objective combinations. The circles indicate seeds. The rewards of the colored shapes are known. The depth of color of the seeds indicates the amount of energy allocated.

1) MPMAB Model Overview: Table I shows the variables in our model. Figure 2 shows the overview of our MPMAB model. First, our MPMAB model handles the problem of adaptive objective combination selection. Each combination stands for a player with his own goal when playing the slot machine. The number of objectives and the number of objective combinations are constant during fuzzing. For instance, if we have 3 objectives to optimize, there are 8 (2^3) objective

combinations in total. Since the number of combinations is constant, the reward of each combination can be captured through a pioneer stage. Then, we choose the best objective combination through our proposed algorithm (Section III-B2).

Second, MPMAB deals with the adaptive energy allocation of seeds under the chosen objective combination. During fuzzing, the number of seeds is increasing, and we cannot estimate the reward until we execute the seed. Therefore, the trade-off between focusing on reward-known seeds (exploitation) and trying reward-unknown seeds (exploration) is not as straightforward as in combination selection. To address this problem, we divide the fuzzing process into exploration and exploitation states. We can adaptively allocate energy in different states under different objective combinations (Section III-B3).

2) Objective Combination Selection: At the $\mathtt{t}$th minute, the ID of the current fuzzing round is $\mathtt{t}$. As the fuzzing campaign proceeds, we can obtain the average value of objective $\mathtt{O_i}$ during round $\mathtt{t}$, which is denoted as $\mathtt{v}^{\mathtt{t}}_{\mathtt{O_i}}$. In addition, we can calculate the average value of objective $\mathtt{O_i}$ in the previous $\mathtt{t}$ rounds (round 1, 2..., $\mathtt{t}$) as

$$\overline{v}^t_{O_i} = \frac{\sum_{k=0}^{t} v^k_{O_i}}{t} \tag{1}$$

Therefore, we define the reward of choosing objective $\mathtt{O_i}$ in round $\mathtt{t}$ as

$$R(O_i, t) = t * (\frac{v^t_{O_i}}{\overline{v}^t_{O_i}} - \lambda * \frac{v^t_{O_0}}{\overline{v}^t_{O_0}}) \tag{2}$$

As we can see from the equation, $\frac{\mathtt{v}^{\mathtt{t}}_{\mathtt{O_i}}}{\overline{\mathtt{v}}^{\mathtt{t}}_{\mathtt{O_i}}}$ denotes the ratio of the objective value in the current round and in the previous $\mathtt{t}$ rounds. When $\mathtt{v}^{\mathtt{t}}_{\mathtt{O_i}}$ is greater than $\overline{\mathtt{v}}^{\mathtt{t}}_{\mathtt{O_i}}$, the reward is large. This encourages selecting the objectives that increase rapidly. Additionally, we emphasize changes in late rounds [23] and multiple $\mathtt{t}$ in front of the reward. In addition, the speed of the fuzzing campaign is crucial to fuzz testing[24], [25], [26]; we add a penalty to objectives that slow down the process: $-\lambda * \frac{\mathtt{v}^{\mathtt{t}}_{\mathtt{O_0}}}{\overline{\mathtt{v}}^{\mathtt{t}}_{\mathtt{O_0}}}$ (speed is the 0th objective).

When the number of objectives is $\mathtt{N}$, the number of objective combinations is $2^{\mathtt{N}}$. We deduce the reward for a combination $\mathtt{C_l}$ as

$$R(C_l, t) = \frac{\sum_{O_i \in C_l} R(O_i, t)}{L} + t * L \tag{3}$$

$\mathtt{L}$ is the number of objectives in this combination. This reward consists of two parts. First, we use the average reward of the objectives. Second, $\mathtt{t} * \mathtt{L}$ is added to reward combinations with more objectives.

Next, we can calculate a final score for the combinations to make decisions. UCB1 [27] is a classic answer to the MAB problem, and we calculate scores based on it as

$$\begin{aligned} Score(C_l, t) &= \overline{R}(C_l, t) + U(C_l, t) \\ &= \frac{\sum_{k=0}^{t} R(C_l, k)}{t} + \gamma * \sqrt{\frac{ln(\sum_{C_l \in C} n_l)}{n_l}} \end{aligned} \tag{4}$$

$\mathtt{C}$ denotes all the objective combinations. The score consists of two parts. $\overline{\mathtt{R}}(\mathtt{C_1}, \mathtt{t})$ is the average reward of $\mathtt{C_1}$ in previous $\mathtt{t}$ rounds, which gives combinations with greater historical rewards higher scores (exploitation). $\mathtt{U}(\mathtt{C_1}, \mathtt{t})$ is the upper confidence bound of $\mathtt{C_1}$, and it adds greater scores to combinations with smaller $\mathtt{n_1}$ values (the number of times the combination is selected), which is exploration. At the beginning of fuzzing, we go through a **pioneer stage**, in which each objective combination is selected once. After this stage, the $\mathtt{n_1}$ value of each combination will be 1. Next, at the end of each round of fuzzing, we calculate the score of each combination and choose the one with the maximum score as the objective combination for the next round. Moreover, γ is a key parameter in UCB1 that controls the balance between exploration and exploitation, and we will discuss this parameter in Section V.

3) Power Schedule: Once we determine the objective combination in this round of fuzzing, our MPMAB model goes to adaptive energy allocation under the chosen objective combination. The number of seeds (arms) increases as the fuzzing campaign continues, and we cannot reuse a pioneer stage to get the reward to start the model. As mentioned above, our key challenge is to balance exploration (trying reward-unknown seeds) and exploitation (choosing the seed with the maximum reward). Our goal is to adaptively assign energy under the chosen objective combination in different fuzzing states. However, according to the related research in energy allocation of CGF including [1], [10], [28], there is no previous work about allocating energy in the situation of multi-objective optimization. Based on this background, we first define the average energy to reach a certain objective value as

$$\overline{E}^t_{O_i} = \frac{Execs(t)}{\overline{v}^t_{O_i}} \tag{5}$$

This is the quotient of the number of executions and the value of an objective in $\mathtt{t}$ rounds. We consider it the **minimum** energy required to increase the objective value.

Likewise, we can deduce the average energy of $\mathtt{C_1}$ in $\mathtt{t}$ rounds as

$$\overline{E}^t_{C_l} = \frac{\sum_{O_i \in C_l} \overline{E}^t_{O_i}}{L} \tag{6}$$

Next, we divide the fuzzing states into different states: **Exploration state.** This state implies that there are currently reward-unknown seeds, and we need to try as many new seeds as possible. We allocate the minimum energy to seeds in this state as

$$E(s_j) = \overline{E}^t_{C_l} \tag{7}$$

This amount of energy has two features: 1) It remains very small. 2) Although it is small, it is the minimum expected energy needed to reach greater objective values. Therefore, we assign this amount of energy to seeds in this state.

Exploitation state. In this state, all the rewards of seeds are known, and it is rational to choose the seed with the maximum reward. We define the energy in this state as

$$E(s_j) = \overline{E}^t_{C_l} * (\frac{v_{C_l}(s_j)}{\overline{v}^t_{C_l}} + \sum_{O_i \in C_l} is\_max(v_{O_i}(s_j))) \tag{8}$$

We can see that Equation 8 is based on the energy in the exploration state in Equation 7. In this equation, $\frac{\mathtt{v_{C_1}(s_j)}}{\overline{\mathtt{v}}^{\mathtt{t}}_{\mathtt{C_1}}}$ is the

ratio between the objective value of executing seed $\mathtt{s_j}$ and the average value. When the objective value of executing the seed ($\mathtt{v_{c_l}(s_j)}$) is greater than the average value ($\overline{\mathtt{v}}^{\mathtt{t}}_{\mathtt{c_l}}$), we allocate more energy to encourage it and vice versa. Additionally, `is_max()` returns 1 if executing this seed reaches the maximum value of a certain objective and returns 0 otherwise. If executing this seed reaches the maximum value of a certain objective, we add a bonus based on this function ($\mathtt{is\_max(v_{o_i}(s_j))}$) to the allocated energy.

C. NIC Algorithm

Algorithm 1 The MobFuzz algorithm.

Require: Initial seeds S, Objectives O
1: $Q = S$
2: **while** $cur\_time < TIME\_OUT$ **do**
3: **if** $cur\_time - prev\_time >= 1\ min$ **then**
4: */* combination selection: obj_com_sel(obj_val, obj_sel, t) */*
5: $Score = \overline{R}(C_l, t) + U(C_l, t)$
6: $C_l = arg\_max(Score)$
7: */* end of combination selection */*
8: $prev\_time = cur\_time$
9: **end if**
10: */* power schedule: pow_sch(Cl, obj_val, t) */*
11: **if** $state == Exploration$ **then** */* current state of fuzzing */*
12: $energy = \overline{E}^t_{C_l}$
13: **end if**
14: **if** $state == Exploitation$ **then**
15: $energy = \overline{E}^t_{C_l} * (\frac{v_{C_l}(s_j)}{\overline{v}^t_{C_l}} + \sum_{O_i \in C_l} is\_max(v_{O_i}(s_j)))$
16: **end if**
17: */* end of power schedule */*
18: **for** $i = 0 \rightarrow energy$ **do** */* energy assigned for each seed */*
19: mut_exe(s) */* seed s is selected by AFL mechanism */*
20: **if** $new\_cov(s) == True$ **then**
21: $Q = Q + s$
22: **end if**
23: **if** $need\_nic == true$ **then**
24: */* NIC: NIC(s', Cl, T) */*
25: **for** $j = 0 \rightarrow T$ **do**
26: cross_mut_exe(s') */* s' is the seed in NIC */*
27: update(O_M)
28: **if** $new\_cov(s') == True$ **then**
29: $Q = Q + s'$
30: **end if**
31: **end for**
32: return $Pareto$ */* Pareto frontier */*
33: */* end of NIC */*
34: **end if**
35: $Q = Q + Pareto$
36: **end for**
37: **end while**
Ensure: Pareto seeds with optimized objectives O_M

1) NIC Overview: To optimize the objectives in the chosen combination and find the optimal result without additional performance overhead, we propose our NIC algorithm. It is an evolutionary algorithm designed for multi-objective optimization in MobFuzz. The basic process of NIC is as follows: First, an initial population of seeds with a scale of `N` is selected. Next, the offspring seeds are obtained with crossover and mutation among the initial population. Second, we execute the target program with each seed in the population and obtain related information (Line 26 in Algorithm 1), e.g., coverage information and objective values. From the second generation onwards, the parent population and the offspring are combined to perform non-dominated sorting [12]. According to the non-dominated relationship of seeds, the seeds with updated objective values (Line 27) are selected to form a new parent population with a scale of `N`. Additionally, the coverage information helps update the seed pool (Lines 28 - 30). Finally, the new offspring seeds are generated by the crossover and mutation among the new parent population. This process repeats until the pre-defined number of iterations is met, and NIC outputs the Pareto frontier, which contains the seeds with the optimal objective values (Line 32).

2) Detailed Techniques of NIC: **Adaptive population size.** The scale of the population is a crucial factor in the NIC algorithm. A small population lacks diversity and cannot produce the optimal result. A large population requires more computing resources in the evolutionary process. Therefore, we need to strike a balance in the size of the population before starting NIC. Through extensive testing, we determine that 10% of the number of seeds should be an appropriate value for the population size. Before we start NIC each time, we randomly select 10% of the seeds from the seed pool to form the initial population, and they go into the evolutionary process.

Co-mutation operators with AFL. We propose 3 techniques for efficient mutation in NIC. First, we integrate the fuzzing-effective operators into NIC, e.g., replacing four bytes with a boundary value of the integer type. Moreover, traditional evolutionary or genetic algorithms retain both offspring to maintain the **diversity** of the population. To address this problem, in NIC, two parent seeds go through crossover and mutation, and we keep both of the generated offspring. Finally, since AFL selects different mutation operators and positions to mutate with equal probability, it cannot highlight the importance of different operators and positions. In addition, in the situation of multi-objective optimization, we need to connect objectives to a specific or some mutation operators. In other words, an objective combination should have a corresponding mutation operator combination, in which the operators should be given a higher probability. To handle these issues, in NIC, we record the number of times the operators and positions can increase the objective values in the chosen objective combination. The ones that are more likely to increase the objective values are selected with a greater probability. In this way, we can get the best mutation operators or positions for a chosen combination.

Reducing performance overhead. The evolutionary or genetic algorithms usually iterate for more than 100 generations. If we directly inserted this process into the fuzzing loop, the performance overhead would be unacceptable. We propose several methods to handle this issue. For example, as described in Section III-C1, when we crossover and mutate the parent population to generate the offspring, we add the process of the new coverage monitor. Seeds generated during NIC that cover new code can also be saved to the seed pool of the main fuzzing loop. Techniques such as this shared seed pool indicate that NIC is no longer independent of the main loop. They are integrated instead. In this way, the overhead to produce the optimal results through the iterations is removed in NIC. In addition, we add a starting condition for NIC. In our design, when we monitor a decrease in the values of the objectives, we start the NIC algorithm to optimize and increase them. Specifically, we record the objective values every minute. When the gradient of two continuous values is less than a threshold, the NIC algorithm is started. According to our preliminary experiments, when the threshold is -0.15, it is the configuration to start NIC to achieve the best performance. Additionally, NIC will execute for a pre-defined number of

iterations (Line 25 in Algorithm 1).

D. Working Flow

In Algorithm 1, the inputs of MobFuzz are the initial seeds `S` and the objectives `O` we want to optimize. The seed pool `Q` is initialized to the user-provided seeds on Line 1. Within the configured timeout period of fuzzing (`TIME_OUT`), MobFuzz continues to fuzz the target program.

To begin, we need to determine the time interval to select the objective combination for the next round. Based on our preliminary evaluation, we take 1 minute as the time interval to make selections. The 24-hour fuzzing campaign is thus divided into 1440 ($\frac{24*60}{1}$) rounds. Each round `t` of fuzzing lasts for 1 minute. At the end of round `t`, we need to select the objective combination for the next round. As shown on Lines 3 - 9, if the time interval exceeds 1 minute, we will adaptively select objective combinations with MPMAB in function `obj_com_sel()`. The arguments of this function include `obj_val` (the values of the objectives), `obj_sel` (the numbers of selections of the objectives), and `t` (the current fuzzing round). This function outputs the selected objective combination C_1. When a combination is determined, in this time interval, we will optimize the objectives in this chosen combination. Section III-B2 shows this procedure in detail.

On Lines 10 - 17 in function `pow_sch()`, we monitor the current state (exploration or exploitation) of fuzzing and assign energy to seeds based on the chosen objective combination. The arguments are listed on Line 10. The output is the energy according to Equation 7 and 8 in Section III-B3. The amount of energy determines the number of mutations and executions on a seed (Line 19). Specifically, we focus on the power schedule. Therefore, we inherit the seed selection mechanism of AFL. After the mutations and executions (`mut_exe(s)`), we save seeds that bring new code coverage (Lines 20 - 22). Lines 24 - 33 show the workflow of NIC. The arguments of `NIC()` include s' (the selected seeds from the pool), C_1 (the chosen objective combination), and `T` (the number of iterations of NIC). Line 26 shows the mutations and executions of NIC. The values of objectives O_M are updated on Line 27. Additionally, the NIC algorithm goes through the evolutionary process with a shared seed pool (Line 29) and optimizes the objective values. The output of NIC is the Pareto frontier [12] which is the set of seeds with the optimal objective values. We add the Pareto frontier as seeds to the seed pool on Line 35. In conclusion, NIC has the following functionalities: 1) It outputs the optimal objective values of the selected objectives. 2) NIC outputs the Pareto frontier, and these seeds are saved to the shared seed pool. 3) At the same time, it saves seeds that bring new coverage.

IV. Implementation

We implement MobFuzz based on AFL. We modify the instrumentation code in LLVM to record the amount of stack memory consumption and number satisfied comparison bytes, respectively. The MPMAB model and the NIC algorithm are implemented separately in `afl-fuzz.c`, and they contain 1.5k lines of code in total. Additionally, we modify the main fuzzing loop to interact with MPMAB and NIC. We replace the original power schedule with the MPMAB schedule, and we add code to check if the starting condition of NIC is met. Specifically, when the gradient of two continuous objective values is less than a threshold, NIC will be started. These modifications contain about 0.5k lines of code.

V. Evaluation

In our evaluation, we answer the following research questions:

- **RQ1.** How does multi-objective optimization in MobFuzz perform compared with single-objective optimization in the baseline fuzzers?
- **RQ2.** How does the objective combination selection adapt in the fuzzing process?
- **RQ3.** How does our power schedule perform compared with the baseline fuzzers under the chosen objective combination?
- **RQ4.** Does NIC optimize the objective values without introducing additional performance overhead?

A. Setup

Target programs to test. We test 12 real-world programs in total. They include programs of various purposes, e.g., image processing (`tiff2pdf`). Table II shows the basic information of these target programs. They are collected from state-of-the-art papers and these papers are listed in Table XVI in Appendix. We believe collecting programs in this way can ensure persuasiveness and representativeness.

TABLE II. Target programs

Targets	Version	Format
avconv -y -i @@ -f null	libav-12.3	mp4
exiv2 @@ /dev/null	exiv2-0.27	jpeg
infototap @@	ncurses-6.1	txt
mp42aac @@ a.aac	Bento4-1.5.1-628	mp4
mp4tag –show-tags –list-symbols –list-keys @@	Bento4-1.5.1-628	mp4
nm -C @@	Binutils-2.30	elf
podofopdfinfo @@	podofo-0.9.6	pdf
readelf -a @@	Binutils-2.30	elf
tiff2pdf @@	libtiff-4.0.7	tiff
tiff2ps @@	libtiff-4.0.7	tiff
podofotextextraxt @@	podofo-0.9.6	pdf
xmllint @@	libxml-2.98	xml

Baseline fuzzers to compare. AFL[7], MemLock[8], and FuzzFactory[20] are used in our evaluation to test real-world programs. According to our discussion in Introduction, they are chosen because we can compare the multi-objective optimization of MobFuzz with the single-objective optimization of them. In addition, we choose 3 objectives as our evaluation metrics, including speed of execution (AFL), stack memory consumption (MemLock), and number of satisfied comparison bytes (FuzzFactory)[1]. These fuzzers use consistent settings: deterministic and havoc.

We use seeds in the `testcase` directory provided by AFL as the initial seeds. Our evaluations are conducted on a server for 10 times and 24 hours.

[1]SP or Speed denotes execution speed, ST or Stack denotes stack memory consumption, and CM or Cmp denotes number of satisfied comparison bytes.

TABLE III. EVALUATION OF MOBFUZZ REGARDING DIFFERENT OBJECTIVES

Targets	Execution speed			Stack memory consumption			Satisfied comparison bytes		
	MobFuzz	AFL	p value/$\hat{A}_{12}$	MobFuzz	MemLock	p value/$\hat{A}_{12}$	MobFuzz	FuzzFactory	p value/$\hat{A}_{12}$
avconv	**85.60**[1]	83.09	0.02/0.77	**58704.00**	37846.40	0.01/0.75	$\mathbf{2.88*10^8}$	$1.51*10^8$	$< 10^{-4}$/1.00
exiv2	**671.41**	372.58	$< 10^{-4}$/1.00	$\mathbf{1.55*10^6}$	$1.21*10^6$	0.01/0.80	$\mathbf{4.66*10^7}$	$2.24*10^7$	$< 10^{-3}$/0.92
infotocap	**386.86**	292.55	$< 10^{-2}$/0.86	**22122.40**	15852.00	0.12/0.60	$\mathbf{2.05*10^8}$	$1.70*10^8$	$< 10^{-2}$/0.90
mp42aac	**1688.83**	1666.99	0.40/0.54	$\mathbf{2.03*10^5}$	$1.81*10^5$	0.02/0.77	$\mathbf{8.34*10^7}$	$4.20*10^7$	$< 10^{-3}$/0.93
mp4tag	**1452.70**	1381.56	0.08/0.69	$\mathbf{2.25*10^5}$	$1.98*10^5$	0.01/0.79	$\mathbf{7.67*10^7}$	$4.23*10^7$	0.01/0.80
nm	**1426.29**	972.73	$< 10^{-3}$/0.94	$\mathbf{8.38*10^6}$	$7.17*10^6$	0.09/0.68	$\mathbf{1.88*10^8}$	$6.57*10^7$	$< 10^{-4}$/1.00
pdfinfo	**875.96**	642.15	$< 10^{-4}$/1.00	3660.00	3660.00	-/0.50	$\mathbf{1.14*10^7}$	$7.29*10^6$	$< 10^{-4}$/1.00
readelf	**1361.56**	1056.57	0.05/0.72	**2663.60**	2382.40	0.43/0.52	$\mathbf{3.08*10^8}$	$7.83*10^7$	$< 10^{-4}$/1.00
tiff2pdf	**1895.09**	1670.86	$< 10^{-4}$/1.00	1588.00	1588.00	-/0.50	$\mathbf{4.38*10^7}$	$3.86*10^7$	0.25/0.49
tiff2ps	**2050.05**	1750.97	$< 10^{-2}$/0.87	1308.00	1308.00	-/0.50	$\mathbf{9.16*10^6}$	$8.48*10^6$	0.03/0.74
txtext	**855.39**	505.29	$< 10^{-4}$/1.00	**3876.10**	3676.00	0.03/0.75	$\mathbf{1.17*10^7}$	$1.12*10^7$	0.05/0.67
xmllint	**1020.47**	746.53	$< 10^{-4}$/1.00	**64112.80**	64085.60	0.03/0.72	$\mathbf{1.65*10^8}$	$6.65*10^7$	$< 10^{-4}$/1.00
Average	**1147.51**	928.48(+23.6%)[2]	0.04/0.87	$\mathbf{8.72*10^5}$	$7.40*10^5$(+17.7%)	0.04/0.66	$\mathbf{9.60*10^7}$	$4.62*10^7$(+107.9%)	0.02/0.87

[1] Greater values are better. [2] The percentages in the brackets of the last line denote the increase in contrast to the baseline fuzzers.

B. Effectiveness of Multi-objective Optimization

1) Results of Objective Values: Table III shows the average objective values of 10 repeated runs. The p values and $\hat{A}_{12}$ values are also listed in the table. The values of MobFuzz are greater than (33 out of 36) or equal to (3 out of 36) the compared values in all the comparisons. Among them, 32 pairs of comparisons show a statistically significant difference ($p < 0.05$ or $\hat{A}_{12} > 0.5$). Specifically, we have 10 comparisons that reach a p value less than 10^{-4} and an $\hat{A}_{12}$ value of 1.0. For example, in xmllint the satisfied comparison bytes of MobFuzz and FuzzFactory are $1.65 * 10^8$ and $6.65 * 10^7$, respectively. MobFuzz is approximately 2x better than that of FuzzFactory. In the `Average` row, we can see that the average value of MobFuzz is greater than that of the baseline fuzzers. In the value of satisfied comparison bytes, we even achieve more than a 100% increase in contrast to FuzzFactory.

MOO is designed to accomplish the task of generating the optimal values for all the objectives. The reason for the improvement against MemLock and FuzzFactory is the NIC algorithm that helps MobFuzz reach the optimal values. NIC keeps looking for the Pareto seeds in an evolutionary process. Every iteration is closer to the optimal values. The final result is closest to the optimal values. MemLock and FuzzFactory have no such mechanism to reach the optimal values for the objectives.

2) Branch Coverage and Unique Bugs: Table IV shows the branch coverage and number of unique bugs found by the fuzzers. Branch coverage is the recommended coverage metric in evaluating fuzzing [29]. In regard to the number of branches, MobFuzz outperforms the baseline fuzzers in 30 out of the 36 comparisons. On average, it can find 6% more branches at most than AFL and FuzzFactory. The unique bugs are manually collected from the unique crashes with the help of AddressSanitizer (ASAN) and gnu debugger (GDB). In the columns of unique bugs, despite the all-zero target programs, MobFuzz outperforms other fuzzers in all the 15 comparisons. The average value of MobFuzz is greater than the competitors, with 3x more bugs compared with AFL and MemLock.

We can see that MobFuzz outperforms the 3 competitors in the comparisons. The reason for this is that MobFuzz has better multi-objective optimization ability than other fuzzers. First, we retain a relatively fast speed in MobFuzz during the fuzzing campaign. This gives MobFuzz more chances to have more coverage and find more unique bugs. Additionally, MobFuzz has greater values in stack memory consumption than the competitors, which can lead to more bugs of the target programs. Third, the number of satisfied comparison bytes of MobFuzz is greater than other fuzzers. Satisfying more comparisons helps in exploring more program branches. Based on these reasons, MobFuzz can have more program coverage and find more unique bugs than the baseline fuzzers.

• *Answer to RQ1: Multi-objective optimization in MobFuzz outperforms every single-objective optimization simultaneously in the baseline fuzzers.*

C. Objective Combination Selection

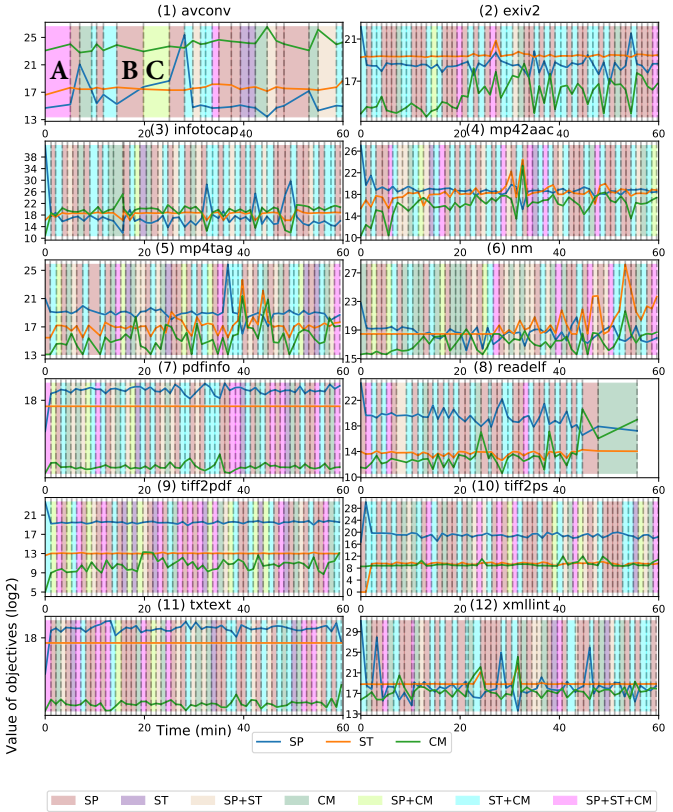

Fig. 3. Values of the objectives and the selected objective combinations within 60 minutes. The 3 lines indicate the values of the objectives. The background colors represent the selected objective combinations within each time interval.

TABLE IV. EVALUATION BASED ON BRANCH COVERAGE AND NUMBER OF UNIQUE BUGS

Targets	Number of edges				Number of unique bugs			
	MobFuzz	AFL	MemLock	FuzzFactory	MobFuzz	AFL	MemLock	FuzzFactory
avconv	**24366.3**[1]	22692.1	23750.8	21676.3	0.0	0.0	0.0	0.0
exiv2	11890.6	11144.7	**11900.1**	11769.1	0.0	0.0	0.0	0.0
infotocap	2319.4	**2421.3**	2351.0	2213.9	**2.8**	0.0	0.0	0.0
mp42aac	**2890.6**	2597.6	2722.7	2747.7	**2.4**	2.0	1.0	0.0
mp4tag	**3081.9**	2733.4	2949.7	2875.8	**2.5**	0.0	1.0	0.0
nm	**7182.1**	7077.5	7041.5	6923.6	0.0	0.0	0.0	0.0
pdfinfo	**2988.3**	2608.3	2608.3	2608.3	0.0	0.0	0.0	0.0
readelf	11444.4	10720.5	**11180.4**	11571.9	0.0	0.0	0.0	0.0
tiff2pdf	1471.6	**1474.6**	1469.2	1471.0	**1.0**	0.0	0.0	0.0
tiff2ps	**451.0**	425.4	425.4	426.0	**1.0**	0.0	0.0	0.0
txtext	**2789.1**	2687.0	2532.0	2599.0	0.0	0.0	0.0	0.0
xmllint	**6072.8**	5878.0	5944.7	5645.0	0.0	0.0	0.0	0.0
Average	**6412.3**	6038.4(6%)[2]	6239.7(3%)	6044.0(6%)	0.8	0.2(300%)	0.2(300%)	0(∞%)

[1] Greater values are better. [2] The percentages in the brackets of the last line denote the increase in contrast to the baseline fuzzers.

1) We set 1 minute as the time interval to make selections. During the 24 hours of the fuzzing campaign, 1,440 selections are made by the MPMAB model in total (possibly less because the target program may still be executing at the end of the time interval). Figure 3 shows how the selected combination affects the values of objectives within each time interval. The background colors show the selected objective combination during this minute. The lines show the values of each objective. We take the result of `avconv` as an example. We mark points `A`, `B` and `C` in the figure of `avconv`. First, point `A` is marked to prove the ability of MobFuzz to optimize multiple objectives simultaneously. In this time interval, the chosen combination is `speed/stack/cmp`. As we can see from the figure, in this round, all the values of objectives increase. Next, at point `B`, `stack/cmp` is selected. We note that as stack and cmp increase, speed decreases, which demonstrates opposite effects between objectives. Finally, point `C` shows our correction to the slowdown of fuzzing. In this time interval, a penalty is added to the stack and cmp objectives because of point `B`, and `speed` is selected to increase the execution speed of fuzzing.

TABLE V. PERCENTAGES OF THE CHOSEN OBJECTIVE COMBINATIONS

Targets	SP	ST/SP/CM	ST/CM	SP/CM	CM	SP/ST	ST
avconv	**63.4%**	3.8%	5.7%	6.5%	6.3%	8.5%	5.8%
exiv2	**86.1%**	4.6%	2.1%	2.1%	1.7%	1.6%	1.8%
infotocap	**83.4%**	4.9%	2.0%	2.2%	3.1%	2.2%	2.1%
mp42aac	**73.0%**	4.0%	3.4%	4.5%	5.1%	4.9%	5.1%
mp4tag	**78.9%**	4.3%	3.5%	3.1%	4.2%	3.5%	2.5%
nm	**77.4%**	4.7%	3.7%	5.4%	3.3%	2.3%	3.2%
pdfinfo	**81.1%**	4.3%	3.1%	3.2%	3.2%	2.6%	2.6%
readelf	**81.5%**	5.1%	2.8%	2.3%	2.1%	3.8%	2.3%
tiff2pdf	**68.3%**	3.7%	4.3%	6.4%	6.2%	5.8%	5.3%
tiff2ps	**83.1%**	4.4%	2.8%	2.0%	2.5%	2.8%	2.5%
txtext	**83.5%**	4.4%	1.8%	3.5%	2.0%	2.8%	2.0%
xmllint	**84.7%**	4.8%	2.8%	2.3%	2.2%	1.5%	1.7%
Average	**78.7%**	4.4%	3.2%	3.6%	3.5%	3.5%	3.1%

2) Table V and Figure 7 (in Appendix) show the distribution of the selected objectives. As we take 3 objectives (speed, stack, and cmp) into consideration, there are 8 objective combinations in total. The most significant observation of Table V and Figure 7 is that the MPMAB model tends to select the speed objective in more than 60% of all the combination selections. Again, we clarify our agreement with previous work [24], [25], [26]: the top priority of fuzzing is execution speed. Therefore, in Equation 2, we add a penalty to objectives that slow down the fuzzing process. It is in line with our tendency toward speed and can also explain the features of Table V and Figure 7.

3) Finally, we need to prove whether we select the best objective combination. Figure 4 shows different strategies in contrast to ours. The `X`-axis is the proportion of speed in all the selected combinations, and different proportions indicate different selection strategies. We set 6 strategies, including 0%, 20%, 40%, 60%, 80%, and 100% proportions of speed. The `Y`-axis shows the ratio of objective values ($\frac{v'}{v_M}$) of different strategies (v') in contrast to our MPMAB selection (v_M). $\frac{v'}{v_M}$ less than `1.0` means the objective value of this selection is less than the MPMAB selection, and this selection is worse. For example, in `avconv`, when the proportion of speed is 40%, we highlight the results with a red dotted line. The speed of this selection strategy is approximately 80% of ours. The stack and cmp are just above 90% of ours.

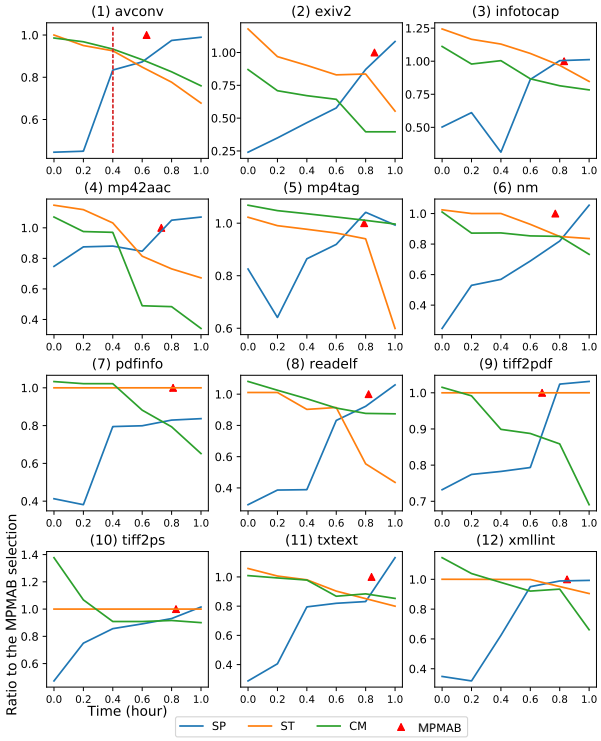

Fig. 4. The `X`-axis denotes different selection strategies. The `Y`-axis is the ratio of objective values ($\frac{v'}{v_M}$) in different selection strategies (v') in contrast to our MPMAB selection (v_M). $\frac{v'}{v_M} < 1.0$ means the objective value of this selection is less than the MPMAB selection and vice versa.

As the proportion of speed increases, the value of speed

TABLE VI. AVERAGE ENERGY CONSUMPTION TO REACH THE OBJECTIVE VALUES OF THE FUZZERS

Targets	Ave. energy of execution speed				Ave. energy of stack memory				Ave. energy of satisfied comparison bytes			
	MobFuzz	AFL	MemLock	FuzzFactory	MobFuzz	AFL	MemLock	FuzzFactory	MobFuzz	AFL	MemLock	FuzzFactory
avconv	**29.33**[1]	39.28	39.06	40.33	**83.45**	90.98	140.22	128.97	**0.01**	0.03	0.03	0.02
exiv2	**22.03**	40.73	37.59	32.22	**10.80**	19.04	16.23	11.01	**0.16**	0.37	0.47	0.29
infotocap	**19.27**	25.51	30.14	30.01	**368.35**	548.58	376.58	502.18	**0.03**	0.08	0.08	0.05
mp42aac	**32.97**	33.32	33.79	34.311	**173.50**	455.01	383.46	420.23	**0.96**	2.75	1.77	1.86
mp4tag	**29.25**	32.90	33.24	33.428	**248.16**	322.99	287.42	358.06	**0.75**	1.61	1.76	1.25
nm	**25.78**	33.18	32.04	32.79	216.21	449.31	277.64	**34.86**	**0.31**	0.66	0.66	0.44
pdfinfo	**31.19**	33.59	33.77	32.91	1.04	**0.73**	0.81	0.73	**1.46**	2.02	3.26	2.72
readelf	**29.88**	33.66	32.61	33.82	**8765.65**	19732.93	9243.49	11510.45	**0.13**	0.81	0.69	0.47
tiff2pdf	**28.95**	33.83	33.15	31.88	**958.27**	2346.60	1117.60	2712.60	**1.84**	1.94	2.23	1.87
tiff2ps	**30.16**	33.27	33.99	32.35	**4.03**	7.38	7.47	7.39	**3.63**	4.75	6.01	5.90
txtext	**29.06**	32.70	33.98	34.39	**0.41**	0.55	0.80	0.66	**1.22**	1.86	3.05	2.96
xmllint	**29.20**	32.64	31.77	33.05	**24.39**	163.85	140.92	260.80	**0.12**	0.46	0.23	0.41
Average	**28.1**	33.7(-17%)[2]	33.8(-17%)	33.5(-16%)	**904.5**	2011.5(-55%)	999.4(-10%)	1329.0(-32%)	**0.87**	1.44(-39%)	1.68(-48%)	1.52(-42%)

[1] Smaller values are better. [2] The percentages in the brackets of the last line denote the decease in contrast to the baseline fuzzers.

also increases. However, as discussed previously in this paper, it is often the case that objectives have opposite effects on each other. The values of stack and cmp decrease as speed increases. More interestingly, we mark the result of MPMAB selection in the figure as a red triangle. The red triangle is close to the intersection of the three lines. In conclusion, none of the 6 selection strategies outperforms the MPMAB selection. The objective values of our strategy are all greater than those of the 6 strategies. The figure indicates that our combination selection method handles the relationship among the multiple objectives, and we can choose the most appropriate proportion of each combination to optimize all the objectives. According to our discussion above, we can answer RQ2.

• *Answer to RQ2: Our selection strategy can adaptively select the best objective combinations.*

D. Power schedule

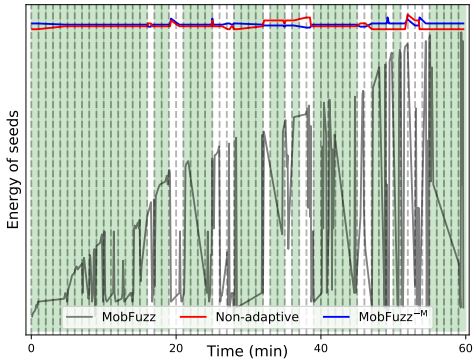

Fig. 5. Comparison of our adaptive power schedule with non-adaptive and MobFuzz^{-M} (MobFuzz without MPMAB) schedules in 1 hour. The Y-axis shows the amount of energy allocated by the schedules. The green background color denotes that in this time interval, our adaptive schedule adjusts the energy according to the chosen objective combination and the non-adaptive schedule fails to.

As described above, there is no previous work solving the problem of energy allocation under the chosen objective combination. To demonstrate that our adaptive power schedule can allocate a different amount of energy to seeds under the chosen objective combination, we show the comparison among our schedule, the non-adaptive (i.e., the power schedule of AFL) schedule, and MobFuzz^{-M} (MobFuzz without MPMAB) schedule in Figure 5. The `X`-axis shows one hour of the fuzzing process. The `Y`-axis shows the energy allocated by our adaptive schedule and others. Moreover, the `X`-axis is divided into 60 minutes, and each minute represents a time interval with different chosen objective combinations. Our schedule adaptively allocates energy according to the chosen combination, where the non-adaptive schedule is insensitive to the changes and keeps allocating the same amount of energy under different objective combinations. Our schedule outperforms the non-adaptive schedule in 43 out of the 60 minutes, which demonstrates the effectiveness of our adaptive power schedule.

In addition, the result of MobFuzz^{-M} schedule is similar to the non-adaptive schedule. This result indicates that without MPMAB, MobFuzz cannot adaptively choose the best objective combination or allocate the appropriate amount of energy. Therefore, there is no change in the allocated energy corresponding to the chosen objective combination. In conclusion, the comparison between MobFuzz and MobFuzz^{-M} proves the effectiveness of MPMAB.

To answer whether our power schedule saves energy, we conduct the following experiments. Table VI shows the average energy consumption to reach the objective values in Table III. We divide it by the number of executions for these objective values and calculate the average energy. As we can see from the table, among all the 108 pairs of comparisons with the baseline fuzzers, only in 4 of them does our power schedule have greater average energy consumption, and these are in `nm` and `pdfinfo`. In other words, MobFuzz allocates less energy for the objectives in over 96% of the scenarios. For instance, in `readelf`, our allocated energy is 6x less (0.13 vs. 0.81) than that of AFL. In the `Average` row of the table, we calculate the averages of all the values. Among all the average values, MobFuzz allocates less energy. Specifically, in comparison with AFL in terms of stack memory, we save more than 50% of energy. From the above discussion, we can answer RQ3.

• *Answer to RQ3: Our power schedule can adaptively allocate energy according to the chosen objective combination and save more energy compared with the baseline fuzzers.*

E. Evaluation on The NIC Algorithm

1) Results of Good Seeds: We define seeds that achieve greater objective values than the average in the selected objective combination as **good seed**s. Figure 6 shows the percentages of good seeds generated in MobFuzz and the baseline fuzzers. We can extract two conclusions from the figure. First, among all the 12 target programs, the percentages of good seeds in MobFuzz are greater than those of the baseline fuzzers. The minimum performance improvement is in `readelf`, which is approximately 2x. As discussed above,

TABLE VII. COMPARISON OF THE MAIN LOOP MUTATION OPERATORS (MAIN) AND NIC MUTATION OPERATORS

Targets	Ave. Time (ms)		Ave. Length (byte)		Num. of executions		Num. of seeds		Executions per seed	
	NIC	Main	NIC	Main	NIC	Main	NIC	Main	NIC	Main
avconv	**10.27**	10.65	**1273.19**	5886.82	$2.00*10^6$	$5.38*10^6$	1796.20	20367.00	1113.46	**264.15**
exiv2	**1.13**	1.26	**259.16**	4140.21	$3.24*10^5$	$5.76*10^7$	74.40	5960.80	**4354.83**	9663.13
infotocap	**1.58**	1.94	**8164.61**	17377.55	$3.20*10^5$	$3.31*10^7$	195.60	5490.30	**1635.99**	6028.81
mp42aac	**0.45**	0.52	**1201.41**	5421.39	88873.40	$5.31*10^8$	53.70	3192.90	**1654.99**	16630.65
mp4tag	**0.44**	0.64	**1230.61**	5570.29	95440.40	$1.24*10^8$	39.80	3061.10	**2398.00**	40508.31
nm	**0.59**	0.62	**1743.88**	5450.27	95241.40	$1.28*10^8$	37.60	3256.10	**2533.01**	39310.83
pdfinfo	**1.06**	1.09	**2605.15**	2652.75	499.10	$7.57*10^7$	0.40	40.30	**1247.75**	$1.88*10^6$
readelf	**0.44**	0.54	**1039.05**	9445.07	$1.60*10^7$	$1.02*10^8$	7671.90	49200.00	2085.53	**2073.17**
tiff2pdf	**0.45**	0.50	**759.55**	4170.77	14367.80	$1.64*10^8$	12.60	1213.00	**1140.30**	$1.35*10^5$
tiff2ps	**0.45**	0.46	**372.43**	4924.48	1472.60	$1.77*10^8$	4.00	398.50	**368.15**	$4.44*10^5$
txtext	**1.20**	1.21	**3289.64**	4360.13	398.00	$6.85*10^7$	5.20	41.00	**76.53**	$1.67*10^6$
xmllint	**0.75**	0.77	**3629.15**	9838.51	$3.92*10^5$	$9.02*10^7$	594.50	6683.40	**658.38**	13496.12
Average	**1.45**	1.54(-5.5%)[1]	**1527.7**	2808.6(-45.6%)	$2.46*10^6$	$4.10*10^7$(P5.7%)[2]	12867.2	88471.6(P12.7%)	**1914.2**	4632.6(-58.7%)

[1] The percentages with a "-" in the brackets of the last line denote the improvements in contrast to Main. [2] The percentages with a "P" denote the proportion of NIC to the total (Main+NIC).

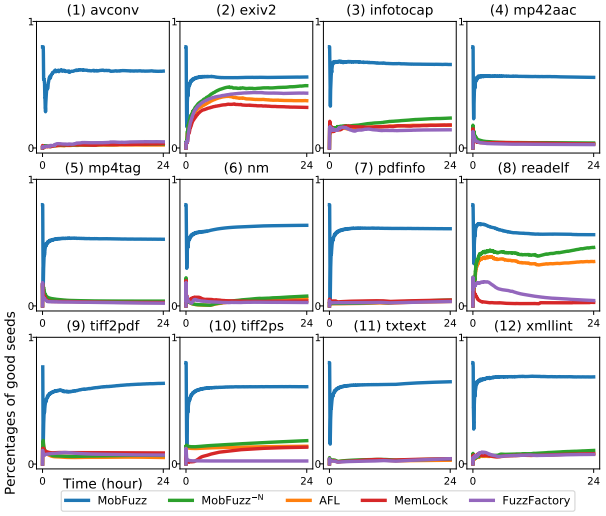

Fig. 6. Percentages of good seeds (seeds that achieve greater objective values than the average in the selected objective combination) generated during fuzzing, where greater values are better. MobFuzz^{-N} denotes MobFuzz without NIC.

in NIC, we optimize the values with crossover, mutation, and execution. The results show that the NIC algorithm in MobFuzz can produce more seeds that are better than the average level, which helps optimize the objectives. Second, without the help of the NIC algorithm, the performance of the baseline fuzzers is similar, which has small percentages of good seeds.

Moreover, we compare MobFuzz with MobFuzz^{-N} (MobFuzz without NIC) in Figure 6. When NIC is disabled, we can see a noticeable difference between MobFuzz and MobFuzz^{-N}. The percentage of good seeds in MobFuzz^{-N} decreases to the level of the baseline fuzzers without NIC. This result indicates that without NIC, MobFuzz^{-N} cannot generate as many good seeds with greater objective values as MobFuzz. In conclusion, by comparing MobFuzz with MobFuzz^{-N}, we demonstrate the effectiveness of NIC.

Along with the results in Figure 5, we can conclude that there is synergy between MPMAB and NIC: 1) The power schedule and NIC are executed under the chosen combination. 2) NIC outputs the Pareto seeds with the optimal values. These values can affect the combination selection and power schedule in return. In this way, these parts can cooperate. The combination of MPMAB and NIC generates the best result.

2) Results of the Mutation Operators: Table VII is the comparison between the mutation operator of NIC and the main fuzzing loop (i.e., the operators of AFL). We can also draw two conclusions based on the table. First, according to the table, in the `Time` and `Length` columns, NIC has less time consumption and shorter length in all of the 24 pairs of comparisons. For the `tiff2ps` target program, NIC is 13x less than Main. Regarding the average values of time and length, NIC also outperforms Main. There is 5.5% less time and 45.6% less length compared with Main. The results demonstrate that with the help of our mutation strategy of selecting operators with better performance according to the chosen objective combination, NIC can achieve better objective optimization.

Second, we can determine the path discovery efficiency of NIC based on the `Executions` and `Seeds` columns. The numbers of executions of Main (i.e., the main loop) and NIC show how many seeds are mutated and executed in each part. The executions of NIC are 5.7% of the total (Main+NIC). If we disable the shared seed pool mechanism, the inputs that cover new paths in NIC will not be saved in the seed pool, and this 5.7% of the executions will be wasted. Moreover, NIC also finds seeds with higher efficiency. NIC generates 12.7% of the total seeds with only 5.7% of the executions. The last two columns show the executions per seed of Main and NIC. These values indicate the number of executions required to find a new seed. We can see a 58.7% decrease in the comparison, which also proves that NIC can generate new seeds more efficiently.

TABLE VIII. PERFORMANCE OVERHEAD OF THE NIC ALGORITHM AND PERFORMANCE IMPROVEMENT OF OUR ADOPTED TECHNIQUES IN NIC

	MobFuzz	[1]MobFuzz^{-N}	[2]MobFuzz^{-POR}
Speed	**1147.51**	1186.52(-3.3%)	1004.07(+14.3%)

[1] MobFuzz^{-N} denotes MobFuzz without NIC.
[2] MobFuzz^{-POR} denotes MobFuzz with NIC but without the performance overhead reduction.

3) Performance Overhead of NIC: Table VIII shows the average execution speed of MobFuzz and two other configurations. MobFuzz^{-N} denotes MobFuzz without the NIC algorithm, and MobFuzz^{-POR} denotes MobFuzz with NIC but without the performance overhead reduction in NIC. First, enabling the NIC algorithm in MobFuzz brings about 3.3% performance overhead to the fuzzing process, which is acceptable. Though NIC slightly slows down the fuzzing process,

it can produce the optimal results for multi-objectives. By sacrificing this 3.3% of speed, we can get optimal results for other objectives. Additionally, even with this 3.3% performance overhead, MobFuzz is still faster than the baseline fuzzers, which is shown in Table III. Second, the comparison with MobFuzz$^{-\text{POR}}$ indicates the performance improvement of the techniques in NIC. The shared seed pool and other techniques increase the speed of fuzzing by 14.3%, which demonstrates the effectiveness of our techniques to reduce the performance overhead in NIC.

TABLE IX. COMPARISON WITH NSGA-II IN THE OBJECTIVE VALUES

	Execution speed	Stack memory	Satisfied comparison bytes
MobFuzz	1147.51	$8.72*10^5$	$9.60*10^7$
[1]MobFuzz$^{\text{N2}}$	327.20	$1.48*10^5$	$8.83*10^6$
[2]MobFuzz$^{\text{N2+APS}}$	559.31	$2.97*10^5$	$1.09*10^7$
[3]MobFuzz$^{\text{N2+CMO}}$	446.89	$6.87*10^5$	$7.63*10^7$
[4]MobFuzz$^{\text{N2+POR}}$	986.37	$3.02*10^5$	$2.56*10^7$

[1] MobFuzz$^{\text{N2}}$ denotes replacing NIC with NSGA-II.
[2] MobFuzz$^{\text{N2+APS}}$ denotes enabling adaptive population size in MobFuzz$^{\text{N2}}$.
[3] MobFuzz$^{\text{N2+CMO}}$ denotes enabling co-mutation operators in MobFuzz$^{\text{N2}}$.
[4] MobFuzz$^{\text{N2+POR}}$ denotes enabling performance overhead reduction in MobFuzz$^{\text{N2}}$.

4) Comparison with Black-box MOO Techniques: To compare with existing black-box MOO techniques such as NSGA-II [12], we replace key design components in MobFuzz with applicable existing designs from NSGA-II and call it MobFuzz$^{\text{N2}}$. Table IX shows the comparison of MobFuzz with MobFuzz$^{\text{N2}}$ in the objective values. In Section III-C, we introduce three aspects of techniques in NIC, including adaptive population size, co-mutation operators, and overhead reduction. None of these techniques is adopted in MobFuzz$^{\text{N2}}$.

MobFuzz$^{\text{N2}}$ uses a fixed initial population size. At the beginning of the fuzzing campaign, this size is too large for the small seed pool. Starting with this large population will slow down the fuzzing process and contribute nothing to increase the objective values. When the seed pool becomes large, this fixed initial population size is not enough. It lacks diversity and cannot produce the optimal result. Therefore, we use adaptive population size to handle these issues. Moreover, by enabling the adaptive population size in MobFuzz$^{\text{N2}}$, we use the result of MobFuzz$^{\text{N2+APS}}$ to demonstrate the effectiveness of this technique. MobFuzz$^{\text{N2+APS}}$ achieves greater values than MobFuzz$^{\text{N2}}$ in the three objectives.

Co-mutation operators are introduced in NIC. The original mutation of NSGA-II does not apply to a fuzzing situation. We remove the ineffective mutation operators in NSGA-II. Moreover, we build connections between objectives and mutation operators. The best operators are selected according to the objectives. In Table IX, the comparison of MobFuzz$^{\text{N2}}$ with MobFuzz$^{\text{N2+CMO}}$ demonstrates the effectiveness of the co-mutation operators. All the objective values are greater in MobFuzz$^{\text{N2+CMO}}$.

In MobFuzz$^{\text{N2}}$, the seeds generated in NSGA-II are independent of the seeds in the main fuzzing loop. Only at the end of the evolutionary process, a small number of seeds are saved to the main seed pool. This independence of the main fuzzing loop wastes the executions during NSGA-II, and it is the primary performance overhead in MobFuzz$^{\text{N2}}$. We propose a shared seed pool technique in MobFuzz to handle this issue. This technique connects the NIC to the main fuzzing loop and saves seeds during the evolutionary process of NIC. In addition, by enabling the performance overhead reduction in MobFuzz$^{\text{N2}}$, the speed result of MobFuzz$^{\text{N2+POR}}$ is much greater than MobFuzz$^{\text{N2}}$. This demonstrates the effectiveness of our overhead reduction techniques.

• *Answer to RQ4: NIC can optimize the objectives without introducing additional performance overhead.*

F. MAGMA Data Set

MAGMA [30] is a newly proposed data set. It contains 7 projects with 19 target programs. MAGMA is a ground-truth fuzzing benchmark based on real programs with real bugs, which allows for a fair and accurate evaluation of fuzzers.

Baseline fuzzers to compare. Following the experiments in the MAGMA paper, AFL[7], AFLFast[1], AFL++[31] (with `lto` and `cmplog` enabled), FairFuzz[17], honggfuzz[32], MOPT [33], and SYMCC [34] are used in our evaluation. The AFL-based fuzzers use consistent settings: deterministic and havoc. Using this configuration can comprehensively test the vulnerability detection ability of MobFuzz compared with state-of-the-art fuzzers.

We use seeds in the `corpus` directory provided by MAGMA as the initial seeds. Our evaluations are conducted on three servers for 10 times and 24 hours.

Experiments in this subsection also follow the configuration in the MAGMA paper, which are divided into two parts: 1) The number of bugs discovered by the fuzzers (the `unique_bugs` results counted by the MAGMA tool script). Table X shows the results of bugs, and Table XI shows the p values of the results. 2) Time to bug (the `TTB` results). Table XII shows the TTB results, and Table XIII shows the p values of the results. Additionally, the TTB results of each bug are in Table XVIII and XIX in Appendix.

TABLE X. AVERAGE NUMBER OF MAGMA BUGS FOUND BY THE FUZZERS

Targets	AFL	AFLFast	AFL++	FairFuzz	honggfuzz	MOPT	SYMCC	MobFuzz
libpng	4.1	4.3	4.2	3.9	4.3	4.0	4.2	4.3
libtiff	8.3	8.0	8.2	6.9	6.0	7.5	6.7	**8.5**
libxml2	6.0	6.1	6.0	6.0	6.5	6.8	5.5	**7.0**
openssl	3.8	3.8	6.5	4.1	6.1	6.0	6.5	6.4
php	6.0	6.5	5.9	5.7	6.2	6.5	5.5	6.2
poppler	7.0	7.0	7.2	7.0	6.5	7.3	6.4	**10.9**
sqlite3	3.8	5.0	2.7	5.0	2.7	1.9	2.0	**5.1**
Average	5.6	5.8	5.8	5.5	5.5	5.7	5.3	**6.9**

1) Bug Count: Table X shows the bugs discovered by the fuzzers, and Table XI shows the p values of these results. According to the results, MobFuzz outperforms all the baseline fuzzers in 4 out of the 7 projects. The average number of bugs of MobFuzz is greater than other fuzzers. Additionally, in `sqlite3`, MobFuzz has the maximum improvement against the baseline fuzzers, which is about 3x better than MOPT. One advantage of MobFuzz over AFL, AFLFast, FairFuzz, honggfuzz, and MOPT is the satisfied comparison bytes MobFuzz concentrates on. By satisfying more bytes, some of the MAGMA bugs can be triggered more easily in MobFuzz. For instance, Bug SQL003 in Table XIX is only triggered when `if(!data)` is satisfied. MobFuzz succeeds in 9 hours, and others fail to.

AFL++ (`cmplog` enabled) and SYMCC can also solve these comparison bytes. However, MobFuzz still outperforms AFL++ and SYMCC in 6 out of the 7 projects. The reason is

TABLE XI. P VALUES OF THE RESULTS IN TABLE X

Targets	AFL	AFLFast	AFL++	FairFuzz	honggfuzz	MOPT	SYMCC	MobFuzz
libpng	$< 10^{-4}$	0.04	$< 10^{-3}$	$< 10^{-3}$	0.01	0.02	0.05	$< 10^{-3}$
libtiff	$< 10^{-5}$	$< 10^{-3}$	0.02	$< 10^{-4}$	0.06	0.45	$< 10^{-4}$	0.01
libxml2	$< 10^{-5}$	$< 10^{-4}$	$< 10^{-3}$	0.06	$< 10^{-4}$	$< 10^{-3}$	0.03	$< 10^{-4}$
openssl	0.03	$< 10^{-4}$	0.08	$< 10^{-3}$	0.01	$< 10^{-3}$	$< 10^{-3}$	$< 10^{-4}$
php	$< 10^{-3}$	0.25	$< 10^{-4}$	0.05	0.02	$< 10^{-3}$	0.22	0.02
poppler	$< 10^{-4}$	$< 10^{-3}$	0.02	0.07	$< 10^{-4}$	0.03	0.01	$< 10^{-5}$
sqlite3	0.07	$< 10^{-3}$	0.02	$< 10^{-5}$	0.27	0.19	$< 10^{-4}$	$< 10^{-4}$

that MobFuzz can optimize other objectives besides the comparison bytes, and AFL++ and SYMCC fail to. MobFuzz can reach a large value of stack memory consumption according to the above experiments. It helps MobFuzz trigger this kind of bug in MAGMA. For example, Bug SQL012 in Table XIX is a stack buffer overflow bug. MobFuzz successfully triggers it in 3.2 hours, and AFL++ and SYMCC fail to.

Additionally, the p values in Table XI of MobFuzz are all less than 0.05. However, some results of others are greater than 0.05, e.g., FairFuzz in `libxml2`. This demonstrates that all the results of MobFuzz are statistically significant.

2) Time to Bug: Table XII shows the TTB results, and Table XIII shows the p values of these results. Detailed TTB results of each bug are in Table XVIII and Table XIX. MobFuzz outperforms all the baseline fuzzers in 4 out of the 7 projects. The maximum performance improvement is in `poppler` compared with AFLFast, which is about 8x. As for the average TTB, MobFuzz has the least TTB compared with others. The reason is that MobFuzz optimizes execution speed and satisfied comparison bytes, and these objectives help to reach less TTB. According to the detailed results, the maximum performance improvement is in Bug PDF007 in Table XIX. MobFuzz solves $\texttt{if}(\texttt{db} \rightarrow \texttt{init.busy})$ and $\texttt{if}(\texttt{zObj} == 0)$ in 2 minutes. AFLFast succeeds in 22.5 hours.

TABLE XII. TTB OF THE FUZZERS

Targets	AFL	AFLFast	AFL++	FairFuzz	honggfuzz	MOPT	SYMCC	MobFuzz
libpng	1.6m[1]	1.7m	1.3m	1.7m	1.5m	1.6m	1.6m	1.3m
libtiff	68.3h[2]	108.0h	55.6h	105.5h	118.1h	69.2h	83.9h	**41.1h**
libxml2	2.2m	2.2m	2.2m	2.2m	53.7m	2.0m	1.8m	**1.6m**
openssl	2.5m	2.5m	**1.6m**	2.5m	2.4m	2.4m	2.4m	1.8m
php	4.9m	6.2m	4.9m	6.8m	5.7m	24.3m	10.1m	**4.0m**
poppler	68.1h	79.3h	48.3h	47.7h	64.7h	27.9h	34.7h	**9.6h**
sqlite3	220.9h	179.3h	**90.7h**	164.4h	126.0h	169.4h	127.3h	107.7h
Average	51.1h	52.4h	26.9h	45.4h	44.3h	38.1h	35.2h	**22.7h**

[1] Smaller values are better. [2] "m" denotes minutes, and "h" denotes hours.

Moreover, the p values in Table XIII of MobFuzz are all less than 0.05. However, some results of other fuzzers are greater than 0.05, e.g., AFL in `libtiff`. This demonstrates that all the TTB results of MobFuzz are statistically significant.

TABLE XIII. P VALUES OF THE RESULTS IN TABLE XII

Targets	AFL	AFLFast	AFL++	FairFuzz	honggfuzz	MOPT	SYMCC	MobFuzz
libpng	0.05	0.42	0.03	$< 10^{-3}$	$< 10^{-4}$	0.02	$< 10^{-3}$	0.03
libtiff	0.12	$< 10^{-3}$	$< 10^{-5}$	$< 10^{-4}$	0.01	$< 10^{-3}$	$< 10^{-4}$	0.01
libxml2	$< 10^{-5}$	0.03	$< 10^{-3}$	0.04	0.01	$< 10^{-4}$	0.29	0.03
openssl	0.33	$< 10^{-4}$	$< 10^{-4}$	0.03	0.01	$< 10^{-5}$	$< 10^{-4}$	$< 10^{-5}$
php	0.03	$< 10^{-4}$	$< 10^{-3}$	$< 10^{-5}$	0.02	0.07	0.01	0.02
poppler	$< 10^{-4}$	0.03	$< 10^{-4}$	0.07	$< 10^{-3}$	0.03	0.01	$< 10^{-5}$
sqlite3	0.06	$< 10^{-4}$	0.02	$< 10^{-4}$	$< 10^{-3}$	0.01	$< 10^{-5}$	0.04

In conclusion, in the MAGMA data set, MobFuzz has better bug detection ability and can find the bugs with less TTB compared with the baseline fuzzers.

VI. DISCUSSION

A. Hyper-parameters

According to our design in Section III-B, we have 2 parameters in our MPMAB model. In Equation 2, λ controls the penalty on objectives that slowdown the fuzzing process. γ in Equation 4 determines the balance between exploration and exploitation. In this section, we study how the parameters affect the performance of MobFuzz.

TABLE XIV. AVERAGE VALUES OF OBJECTIVES OF THE 12 TARGET PROGRAMS WITH DIFFERENT VALUES OF λ

Values of λ	Speed	Stack	Cmp
0.00	998.01	$1.02 * 10^6$	$9.96 * 10^7$
0.01	1019.12	$9.10 * 10^5$	$9.71 * 10^7$
0.10	1147.51	$8.72 * 10^5$	$9.60 * 10^7$
1.00	1150.81	$6.78 * 10^5$	$2.38 * 10^7$
10.00	1155.22	$3.66 * 10^5$	$8.51 * 10^6$

Table XIV shows the average values of objectives of the 12 target programs with different values of λ. We choose 5 values of λ to show the effect on the performance of MobFuzz. In the `Speed` column, the execution speed of fuzzing increases as λ increases. Ideally, we may choose the fastest configuration ($\lambda = 10.00$) because we prefer speed in fuzzing as discussed above. However, when the value of λ is greater than 0.10, the values of stack memory consumption and number of satisfied comparison bytes decrease rapidly, with approximately a 2x decrease in the `Stack` column and a 12x decrease in the `Cmp` column. Therefore, we balance the values of the objectives and choose 0.10 as our configuration of λ.

TABLE XV. AVERAGE VALUES OF OBJECTIVES OF THE 12 TARGET PROGRAMS WITH DIFFERENT VALUES OF γ

Values of γ	Speed	Stack	Cmp
0.00	512.89	$1.44 * 10^6$	$2.09 * 10^8$
0.01	1147.51	$8.72 * 10^5$	$9.60 * 10^7$
0.10	1101.97	$7.21 * 10^5$	$7.55 * 10^7$
1.00	1122.09	$7.96 * 10^5$	$5.63 * 10^7$
10.00	1183.10	$4.61 * 10^5$	$1.07 * 10^7$

Table XV shows the average values of objectives of the 12 target programs with different values of γ. This parameter controls the balance between exploration and exploitation in Equation 4. Greater γ means more exploration, and less means more exploitation. We study 5 different values of γ and how γ affects the performance of MobFuzz. When γ is set to 0, the model considers only exploitation, and objectives with greater historical values will be assigned greater scores according to Equation 4. In this situation, the number of satisfied comparison bytes will get the greatest score since the `Cmp` values are greater than other objective values. Therefore, it can reach the value of $2.09 * 10^8$. As γ increases, there will be more exploration. Objectives with smaller values will be assigned greater scores as γ increases. We study different values of γ and the values of the objectives. We finally choose 0.01 as the configuration. This configuration can produce fast execution speed with appropriate values of stack memory consumption and number of satisfied comparison bytes.

B. Moving to More Objectives

We choose execution speed, stack memory consumption, and number of satisfied comparison bytes as our objectives in this paper, which is not to say that MobFuzz can handle only three objectives. MobFuzz can be extended to optimize more than 3 objectives with minor changes. For example, if we want to add the number of vulnerable function calls as the 4th objective in MobFuzz, we need to instrument the source code

to record the vulnerable functions. Then, we need to modify the number of objectives in the MobFuzz configuration. No further modification is required to optimize these 4 objectives.

C. Threats to Validation

The randomness in fuzzing is the major threats to validation [1], [10], [35], [17]. To solve this problem, we conduct repeated experiments to calculate the average values. The p values and $\hat{A}_{12}$ values are given in our experiments to prove the statistically significant difference. In addition, certain setups of our experiments can be slightly improved. For example, we only set 6 comparison selection strategies in Figure 4. More strategies could be introduced in the experiments to enrich the comparisons.

VII. Related Work

A. MAB Model in Fuzzing

The MAB model deals with the problem of optimizing the total reward in finite trials when we are making choices. In fuzzing, there are many situations where we need to maximize the reward. For example, the ultimate goal of fuzzing is to expose as many bugs as possible. Woo et al. [36] modeled the parameter configuration as the MAB problem to find more bugs. However, in CGF, the idea of allocating more energy to arms with more bugs will result in triggering the same bugs. Therefore, the number of bugs is not included in our objectives in MobFuzz.

Moreover, Patil et al. [37] formalized the process of assigning executions to a test case (energy) as a contextual bandit problem. They proposed a learned model through the policy gradient method to control the energy. Yue et al. [10] improved this model and proposed a variant of the adversarial MAB (VAMAB) model. They explained the details of the fuzzing process as a VAMAB model and considered the balance between exploration and exploitation thoroughly. However, in the situation of multiple objectives, we propose our MPMAB model. In contrast to previous work, our MPMAB model deals with the problem of multiple selections combined together. When we have more than one decision to make, e.g., objective combination selection and energy allocation, the classic MAB model is inadequate. Our MLMAB model makes progress in these multiple-selection scenarios.

B. MOO in Fuzzing

According to our investigation, there are three existing fuzzing tools dealing with multi-objectives in fuzzing. Cerebro uses the idea of the Pareto frontier and non-dominated sorting in [12], as does our NIC algorithm. There are differences between Cerebro and MobFuzz. First, Cerebro does not select objective combinations. As discussed above, there are internal relationships among objectives, which requires us to select the most proper objectives in the current situation. The second difference is whether the optimizing process is an evolutionary procedure. In Cerebro, seeds go through non-dominated sorting, and the Pareto frontier is calculated, which is currently the optimal result. However, this process is only executed once in a fuzzing cycle. We argue that it cannot produce the global optimal solution. Calculating the Pareto frontier and reaching convergence usually require more than 100 iterations through the evolutionary process[12]. Additionally, MOOFuzz [38] has a similar idea to that of Cerebro. Therefore, we argue that it also cannot produce the global optimal solution.

In contrast to MOO in existing fuzzers, MobFuzz selects objective combinations according to the fuzzing state and integrates the evolutionary process into fuzzing without introducing additional overhead according to our evaluation. Based on this, we can produce the global optimal result for multiple objectives without incurring wasted time. In regard to FuzzFactory (two-objective mode) [20], it is more naive when handling multiple objectives. It uses two continuous `if` statements to determine which is better, which can be problematic. For example, it compares seed `X` with `Y` by $\texttt{if}(\texttt{A}_\texttt{X} > \texttt{A}_\texttt{Y})\{\texttt{if}(\texttt{B}_\texttt{X} > \texttt{B}_\texttt{Y})\{\texttt{prefer X}\}\}$ or $\texttt{if}(\texttt{B}_\texttt{X} > \texttt{B}_\texttt{Y})\{\texttt{if}(\texttt{A}_\texttt{X} > \texttt{A}_\texttt{Y})\{\texttt{prefer X}\}\}$. Putting objective `A` in front of `B` or vice versa will lead to incorrect objective optimization. In contrast, MobFuzz produces the optimal value through an evolutionary process and will not fall prey to the above incorrect situations.

C. Power Schedule

The power schedule of CGF controls the number of mutations and executions on a seed. The original power schedule of AFL allocates more energy than is needed[1], [10]. AFLFast [1] was the pioneering work in improving the power schedule of AFL. It uses a transition probability model to describe the relationship between program paths. AFLFast reduces the energy consumption of AFL through a power schedule and search strategy. Later, Yue et al. proposed EcoFuzz [10] to describe the details in the transitions of paths and rewards of seeds through a variant of the adversarial MAB model. Based on the model, the fuzzing process is divided into different states, and different energy values are allocated in these states. Additionally, Entropic [28] proposes an entropy-based power schedule to allocate more energy to seeds with more information. In contrast, MobFuzz utilizes the MPMAB model to solve two problems, including energy allocation. First, objective combinations are selected. Next, it adaptively allocates appropriate energy to seeds based on the selected combination. Our key contribution is that we extend the power schedule beyond the scope of path coverage. Both AFLFast and EcoFuzz emphasize the path discovery ability of seeds in energy allocation, and information entropy in Entropic is also related to coverage. We design the power schedule to allocate energy based on the objectives we selected. MobFuzz can adjust energy more adaptively based on the current objectives, which broadens the application scenarios of the power schedule in CGF, and this is the key difference compared with previous work.

D. Seed Selection

In the fuzzing campaign, the fuzzer needs to choose a seed to fuzz when the previous round of fuzzing is finished. It is important to select the best seed in the seed pool based on the goal of the fuzzer. For example, AFL prefers a seed with a faster execution speed and shorter length. When a seed is marked as favored, it will be selected with a higher probability in the next round. Following AFL, MemLock [8] and FuzzFactory [20] prefer seeds with more memory consumption and more satisfied comparison bytes, respectively. However,

both of them try to optimize only one objective when selecting seeds. As discussed above, there are many situations in which we need to optimize multiple objectives to find deeper bugs. Using only one objective in fuzzing cannot reach the bugs that require multiple triggering conditions. In contrast, we note the lack of consideration and mis-handling of the multiple objectives in CGF. Based on this, we design the MPMAB model to adaptively select the objectives and allocate energy, and we optimize the objectives with NIC. Furthermore, AFLGo [39] and CollAFL [35] also select seeds with their specific goals. However, they require complex program analysis to finish the task. Unlike them, MobFuzz does not need additional static analysis to optimize the objectives. IJON [40] proposes an annotation mechanism to help the analysts select seeds and guide the fuzzing process. Compared with it, MobFuzz is an automatic fuzzing tool that requires no manual effort to guide the fuzzing process.

VIII. Conclusion

In this paper, we propose MobFuzz to handle the problem of multi-objective optimization in gray-box fuzzing. In MobFuzz, we design a multi-player multi-armed bandit model to adaptively select the objective combinations and allocate energy to seeds. We also propose the NIC algorithm to optimize the objectives without incurring additional performance overhead. Based on the experiments on real-world target programs and the MAGMA data set, we demonstrate the improvements in MobFuzz in contrast to the baseline fuzzers.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments and helpful suggestions. This work is supported by the Natural Science Foundation of China (61902412, 61902405, and 61902416), the Research Project of National University of Defense Technology (ZK20-17 and ZK20-09), the Natural Science Foundation of Hunan Province of China (2021JJ40692), and the National High-level Personnel for Defense Technology Program (2017-JCJQ-ZQ-013).

References

[1] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[2] W. Miller and D.L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, SE-2(3):223–226, 1976.

[3] Phil McMinn. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163. IEEE, 2011.

[4] Mark Harman, Yue Jia, and Yuanyuan Zhang. Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–12, 2015.

[5] Xin Yao. Some recent work on multi-objective approaches to search-based software engineering. In Günther Ruhe and Yuanyuan Zhang, editors, *Search Based Software Engineering*, pages 4–15, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[6] Kalyanmoy Deb. Multi-objective optimization. In *Search methodologies*, pages 403–449. Springer, 2014.

[7] Michal Zalewski. American fuzzy lop. In */url-https://lcamtuf.coredump.cx/afl/*, 2013.

[8] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. Memlock: Memory usage guided fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 765–777, New York, NY, USA, 2020. Association for Computing Machinery.

[9] Michal Zalewski. American fuzzy lop technical details. In */url-https://lcamtuf.coredump.cx/afl/technical_datails.txt*, 2013.

[10] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2307–2324. USENIX Association, August 2020.

[11] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. Cerebro: Context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 533–544, New York, NY, USA, 2019. Association for Computing Machinery.

[12] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[13] P. Whittle. Multi-armed bandits and the gittins index. *Journal of the Royal Statistical Society. Series B (Methodological)*, 42(2):143–149, 1980.

[14] J. Wang, C. Song, and H. Yin. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. In *Network and Distributed System Security Symposium*, 2021.

[15] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers, 2020.

[16] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. NDSS, 2020.

[17] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485, 2018.

[18] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min. Ptfuzz: Guided fuzzing with processor trace feedback. *IEEE Access*, 6:37302–37313, 2018.

[19] Michal Zalewski. Afl vulnerability trophy case. In */url-https://lcamtuf.coredump.cx/afl/bugs*, 2013.

[20] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. Fuzzfactory: Domain-specific fuzzing with waypoints. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[21] Sébastien Bubeck and Nicolò Cesa-Bianchi. Regret analysis of stochastic and nonstochastic multi-armed bandit problems, 2012.

[22] Ketan Patil and Aditya Kanade. Greybox fuzzing as a contextual bandits problem, 2018.

[23] Jinghan Wang, Chengyu Song, and Heng Yin. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. 01 2021.

[24] Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, and Yu Jiang. Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 858–870, 2020.

[25] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 787–802. IEEE, 2019.

[26] Cao Zhang, Wei Yu Dong, and Yu Zhu Ren. Instrcr: Lightweight instrumentation optimization based on coverage-guided fuzz testing. In *2019 IEEE 2nd International Conference on Computer and Communication Engineering Technology (CCET)*, pages 74–78. IEEE, 2019.

[27] Rajeev Agrawal. Sample mean based index policies with o (log n) regret for the multi-armed bandit problem. *Advances in Applied Probability*, pages 1054–1078, 1995.

[28] Marcel Böhme, Valentin JM Manès, and Sang Kil Cha. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference*

and Symposium on the Foundations of Software Engineering, pages 678–689, 2020.

[29] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[30] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(3):1–29, 2020.

[31] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: Combining incremental steps of fuzzing research. In *14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20)*, 2020.

[32] Google. honggfuzz: Security oriented software fuzzer. supports evolutionary, feedback-driven fuzzing based on code coverage (sw and hw based). In */urlhttps://honggfuzz.dev/*, 2015.

[33] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, Santa Clara, CA, August 2019. USENIX Association.

[34] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with symcc: Don't interpret, compile! In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 181–198, 2020.

[35] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696, 2018.

[36] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer Communications Security*, CCS '13, page 511–522, New York, NY, USA, 2013. Association for Computing Machinery.

[37] Ketan Patil and Aditya Kanade. Greybox fuzzing as a contextual bandits problem. *CoRR*, abs/1806.03806, 2018.

[38] Xiaoqi Zhao, Haipeng Qu, Wenjie Lv, Shuo Li, and Jianliang Xu. Moofuzz: Many-objective optimization seed schedule for fuzzer. *Mathematics*, 9(3), 2021.

[39] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery.

[40] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1597–1612. IEEE, 2020.

APPENDIX

A. Evaluation of the MPMAB Model

TABLE XVI. 12 REAL-WORLD PROGRAMS AND RELATED STATE-OF-THE-ART PAPERS

Programs	Papers and publication
avconv	AFLSMART (TSE2019), MOPT (Security2019), Intriguer (CCS2019)
exiv2	CollAFL (S&P2018), SLF (ICSE2019), UNIFUZZ (Security2021)
infotocap	MOPT (Security2019), EcoFuzz (Security2020), UNIFUZZ (Security2021)
mp42aac	MOPT (Security2019), MemLock (ICSE2020), UNIFUZZ (Security2021)
mp4tag	MemLock (ICSE2020), PANGOLIN (S&P2020), Fuzzguard (Security2020
nm	Angora (S&P2018), FuZZan (Security2020), EcoFuzz (Security2020)
pdfinfo	MOPT (Security2019), ProFuzzer (S&P2019), Fuzzguard (Security2020)
readelf	Fuzzification (Security2019), MEUZZ (RAID2020), OptiMin (ISSTA2021)
tiff2pdf	Steelix (FSE2017), GREYONE (Security2020), Intriguer (CCS2019)
tiff2ps	QSYM (Security2018), Matryoshka (CCS2019), MEUZZ (RAID2020)
txtext	ProFuzzer (S&P2019), Fuzzguard (Security2020), Ferry (Security2022)
xmllint	AFLFast (CCS2016), Matryoshka (CCS2019), EcoFuzz (Security2020)

Table XVI lists the 12 real-world programs used in our experiments and the related papers which used the programs.

Objective combination selection. Figure 7 shows the percentages of the chosen objective combinations. Different colors indicate different objective combinations.

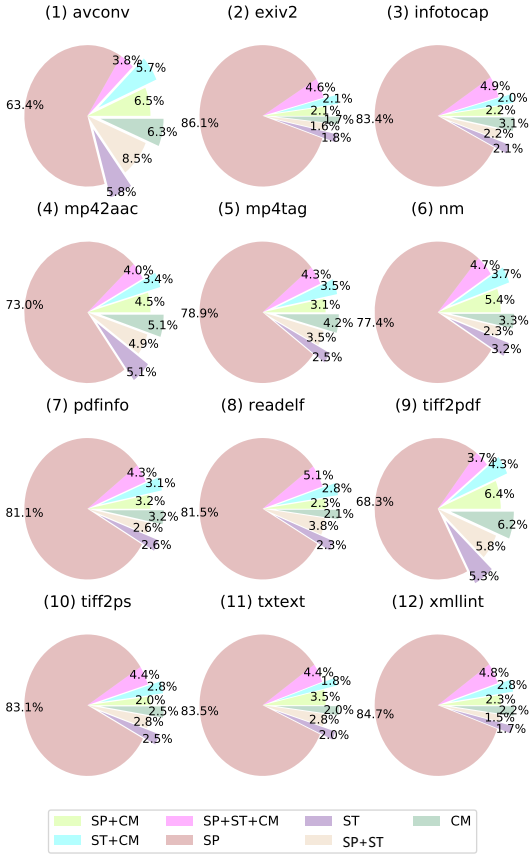

Fig. 7. Percentages of the chosen objective combinations. Different colors indicate different objective combinations.

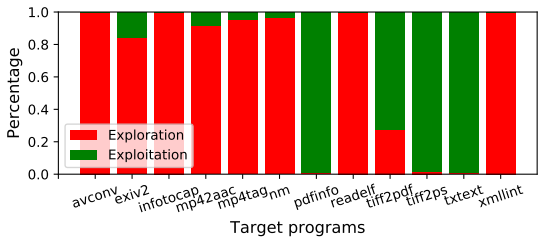

Fig. 8. Percentages of exploration and exploitation states during fuzzing.

Power Schedule. Figure 8 demonstrates the percentages of exploration and exploitation states during fuzzing. Exploration and exploitation can transfer to each other depending on whether new seeds are generated. As we can see from the figure, the percentages of different target programs are completely different. The number of total paths of the programs differs. Therefore, the transformations between exploration and exploitation are different. MobFuzz discovers new paths more easily in some programs, and the state transfers to the exploration state. If no new seeds are generated, the exploitation state begins.

B. Evaluation on NIC

We propose our new strategy by selecting the mutation operators with better historical performance. Figure 9 shows the proportion of the selected operators. We can see that

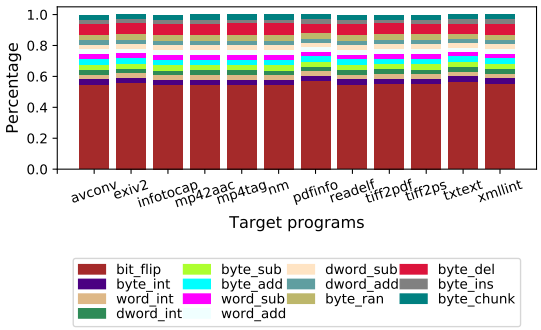

Fig. 9. Percentages of mutation operators during fuzzing.

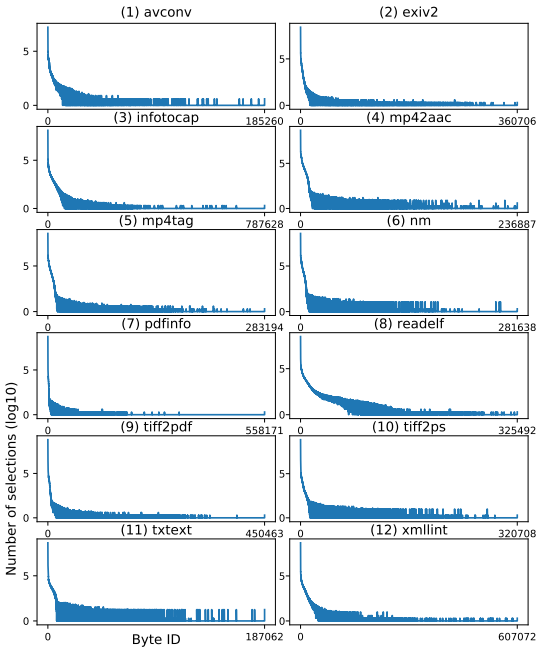

Fig. 10. The number of times a byte is selected.

flipping 1 bit of the seed covers more than 50% of the figure, which demonstrates the simplicity and effectiveness of the `bitflip` operator.

Figure 10 shows the number of selections on the bytes of seeds. (10, 999) indicates that the 10th byte of the input is selected 999 times by the mutation operator. We also tend to select the bytes with better historical performance in our design. In the figure, bytes in the front positions of the seed are more likely to be chosen, which implies that mutating these bytes can bring more reward to the fuzzing process. For example, the header of a JPEG file contains structural information about the file. The header bytes are more important than the subsequent bytes.

C. Bugs in the 12 Real-world Programs

Table XVII shows the real bugs discovered by MobFuzz in the 12 real-world programs. These bugs are collected from the unique crashes with the help of ASAN and GDB. In this table, most of the bugs are overflow bugs. From this result, we can see the vulnerability detection ability of MobFuzz.

TABLE XVII. BUGS DISCOVERED BY MOBFUZZ IN THE 12 REAL-WORLD PROGRAMS

Programs	Bug description
infotocap	heap-buffer-overflow in captoinfo.c in _nc_infotocap()
infotocap	global-buffer-overflow in comp_hash.c in _nc_find_entry()
infotocap	stack-buffer-overflow in infotocap
mp42aac	heap-buffer-overflow in Ap4TrunAtom.cpp in AP4_TrunAtom()
mp42aac	heap-buffer-overflow in Ap4Utils.cpp in SkipBits()
mp42aac	heap-buffer-overflow in Ap4Dec3Atom.cpp in AP4_Dec3Atom()
mp4tag	heap-buffer-overflow in Ap4ByteStream.cpp in WritePartial()
mp4tag	heap-buffer-overflow in Ap4RtpAtom.cpp in AP4_RtpAtom()
mp4tag	heap-buffer-overflow in Ap4Utils.h in AP4_BytesToUInt32BE
tiff2pdf	heap-buffer-overflow in t2p_read_tiff_size()
tiff2ps	heap-buffer-overflow in tiff2ps.c in PSDataColorContig()

Moreover, MobFuzz can detect more unique bugs than the baseline fuzzers.

D. Detailed TTB in the MAGMA Data Set

Table XVIII and Table XIX show the detailed TTB of each bug in the MAGMA data set. In total, there are 118 bugs in the data set. The identification of each bug is in the format of "Project+Number", e.g., "PNG001" denotes the 1st bug in the libpng project. In addition, "PNG" denotes libpng, "TIF" denotes libtiff, "XML" denotes libxml2, "SSL" denotes openssl, "PHP" denotes php, "PDF" denotes poppler, and "SQL" denotes splite3.

TABLE XVIII. DETAILED TTB IN THE MAGMA DATA SET

Bug ID	AFL	AFLFast	AFL++	FairFuzz	honggfuzz	MOPT	SYMCC	MobFuzz
PNG001	15.0s[1]	15.0s	10.0s	15.0s	15.0s	15.0s	15.0s	15.0s
PNG002	×[2]	×	×	×	×	×	×	×
PNG003	20.0s	20.0s	15.0s	20.0s	15.0s	15.0s	20.0s	10.0s
PNG004	15.0s	20.0s	15.0s	20.0s	20.0s	20.0s	20.0s	15.0s
PNG005	15.0s	15.0s	15.0s	15.0s	15.0s	15.0s	15.0s	10.0s
PNG006	15.0s	15.0s	10.0s	15.0s	10.0s	15.0s	10.0s	15.0s
PNG007	15.0s	15.0s	15.0s	15.0s	15.0s	15.0s	15.0s	10.0s
TIF001	23.3h	21.9h	×	×	×	×	×	4.2h
TIF002	7.0m	×	5.0h	8.3h	×	4.5h	13.6h	2.2h
TIF003	2.0m	15.0s	15.0s	15.0s	15.0s	45.0s	15.0s	10.0s
TIF004	×	×	×	×	×	×	×	×
TIF005	×	×	180	×	180	×	×	×
TIF006	9.0h	×	4.0m	×	×	11.8h	12.7h	3.0m
TIF007	1.0m	50.0s	15.0s	1.0m	15.0s	15.0s	15.0s	15.0s
TIF008	3.8h	6.0h	×	×	×	4.0h	7.1h	3.1h
TIF009	6.7h	7.0h	2.4h	1.0h	22.5h	2.0h	2.1h	8.3h
TIF010	1.1h	1.0h	29.0m	27.0m	5.0m	13.0m	29.2m	4.0m
TIF011	×	×	×	×	×	×	×	×
TIF012	10.0s	10.0s	10.0s	10.0s	10.0s	10.0s	10.0s	5.0s
TIF013	×	×	×	×	×	×	×	×
TIF014	1.0m	50.0s	15.0s	1.0m	15.0s	15.0s	15.0s	15.0s
XML001	15.0s	15.0s	20.0s	15.0s	10.0s	15.0s	10.0s	15.0s
XML002	×	×	×	×	×	×	×	×
XML003	15.0s	15.0s	10.0s	15.0s	15.0s	15.0s	15.0s	15.0s
XML004	×	×	×	×	×	×	×	×
XML005	×	×	×	×	×	×	×	×
XML006	15.0s	15.0s	20.0s	15.0s	20.0s	15.0s	15.0s	10.0s
XML007	×	×	×	×	×	×	×	×
XML008	22.0s	20.0s	15.0s	20.0s	15.0s	10.0s	10.0s	15.0s
XML009	15.0s	15.0s	10.0s	15.0s	15.0s	15.0s	15.0s	5.0s
XML010	×	×	×	×	×	×	×	×
XML011	15.0s	15.0s	20.0s	15.0s	52.0m	15.0s	15.0s	10.0s
XML012	15.0s	15.0s	20.0s	15.0s	10.0s	15.0s	15.0s	15.0s
XML013	×	×	×	×	×	×	×	×
XML014	×	×	×	×	×	×	×	×
XML015	×	×	×	×	×	×	×	×
XML016	×	×	×	×	×	×	×	×
XML017	20.0s	20.0s	15.0s	20.0s	15.0s	20.0s	15.0s	10.0s
XML018	×	×	×	×	×	×	×	×
SSL001	20.0s	20.0s	15.0s	20.0s	20.0s	20.0s	20.0s	20.0s
SSL002	15.0s	15.0s	15.0s	15.0s	15.0s	15.0s	15.0s	15.0s
SSL003	20.0s	20.0s	20.0s	20.0s	15.0s	15.0s	15.0s	10.0s
SSL004	×	×	×	×	×	×	×	×
SSL005	15.0s	15.0s	15.0s	15.0s	15.0s	15.0s	15.0s	5.0s
SSL006	×	×	×	×	×	×	×	×
SSL007	×	×	×	×	×	×	×	×
SSL008	15.0s	15.0s	20.0s	15.0s	15.0s	15.0s	15.0s	10.0s
SSL009	15.0s	15.0s	15.0s	15.0s	10.0s	15.0s	15.0s	15.0s
SSL010	10.0s	10.0s	10.0s	10.0s	10.0s	10.0s	10.0s	5.0s
SSL011	×	×	×	×	×	×	×	×
SSL012	×	×	×	×	×	×	×	×
SSL013	×	×	×	×	×	×	×	×
SSL014	×	×	×	×	×	×	×	×
SSL015	×	×	×	×	×	×	×	×
SSL016	15.0s	15.0s	15.0s	15.0s	15.0s	15.0s	15.0s	10.0s
SSL017	×	×	×	×	×	×	×	×
SSL018	×	×	×	×	×	×	×	×
SSL019	10.0s	10.0s	10.0s	10.0s	10.0s	10.0s	10.0s	10.0s
SSL020	15.0s	15.0s	15.0s	15.0s	20.0s	15.0s	15.0s	10.0s
SSL021	×	×	×	×	×	×	×	×
SSL022	×	×	×	×	×	×	×	×

[1] "s" denotes seconds, "m" denotes minutes, and "h" denotes hours. [2] "×" denotes the fuzzer cannot find the bug.

TABLE XIX. DETAILED TTB IN THE MAGMA DATA SET

Bug ID	AFL	AFLFast	AFL++	FairFuzz	honggfuzz	MOPT	SYMCC	MobFuzz
PHP001	×	×	×	×	×	×	×	×
PHP002	15.0s	15.0s	10.0s	15.0s	10.0s	15.0s	15.0s	10.0s
PHP003	15.0s	15.0s	15.0s	15.0s	15.0s	15.0s	15.0s	15.0s
PHP004	15.0s	15.0s	15.0s	15.0s	10.0s	15.0s	15.0s	5.0s
PHP005	×	×	×	×	×	×	×	×
PHP006	15.0s	17.0s	15.0s	10.0s	10.0s	20.0s	10.0s	10.0s
PHP007	×	×	×	×	×	×	×	×
PHP008	×	×	×	×	×	×	×	×
PHP009	15.0s	15.0s	15.0s	15.0s	15.0s	15.0s	15.0s	10.0s
PHP010	×	×	×	×	×	×	×	×
PHP011	10.0s	10.0s	15.0s	10.0s	10.0s	10.0s	10.0s	10.0s
PHP012	×	×	×	×	×	×	×	×
PHP013	×	×	×	×	×	×	×	×
PHP014	×	×	×	×	×	×	×	×
PHP015	3.0m	4.0m	3.0m	5.0m	4.0m	22.0m	8.0m	2.5m
PHP016	×	×	×	×	×	×	×	×
PHP017	10.0s	15.0s	10.0s	10.0s	10.0s	15.0s	15.0s	15.0s
PHP018	×	×	×	×	×	×	×	×
PHP019	×	×	×	×	×	×	×	×
PHP020	10.0s	15.0s	10.0s	10.0s	10.0s	15.0s	15.0s	10.0s
PHP021	10.0s	15.0s	10.0s	10.0s	10.0s	15.0s	15.0s	5.0s
PDF001	40.0s	35.0s	20.0s	35.0s	55.0s	35.0s	25.0s	15.0s
PDF002	3.3h	22.0m	48.0m	2.9h	15.2h	2.0m	1.9h	5.5h
PDF003	20.0s	15.0s	20.0s	15.0s	25.0s	15.0s	15.0s	10.0s
PDF004	×	×	×	×	×	×	×	×
PDF005	25.0s	20.0s	25.0s	20.0s	60.0s	20.0s	20.0s	15.0s
PDF006	30.0s	25.0s	30.0s	25.0s	20.0s	25.0s	20.0s	25.0s
PDF007	2.0h	22.5h	45.0m	3.0h	2.5m	4.0m	1.3h	2.0m
PDF008	25.0s	25.0s	15.0s	25.0s	25.0s	25.0s	20.0s	25.0s
PDF009	5.1h	55.0m	41.0m	40.0m	32.0m	50.0m	1.1h	30.0m
PDF010	7.1h	4.8h	1.0h	3.2h	×	41.0m	1.3h	29.0m
PDF011	15.0s	15.0s	15.0s	15.0s	15.0s	15.0s	15.0s	10.0s
PDF012	15.0s	15.0s	15.0s	15.0s	15.0s	15.0s	15.0s	15.0s
PDF013	×	×	×	×	×	×	×	×
PDF014	20.8h	21.1h	3.2h	7.0h	51.0h	14.7h	3.3h	28.0m
PDF015	×	×	×	×	×	×	×	×
PDF016	3.6h	1.0h	1.0h	3.4h	3.3m	4.0m	41.7m	2.0h
PDF017	×	×	×	×	×	×	×	×
PDF018	×	×	11.5h	×	×	12.0h	×	1.0h
PDF019	4.4h	6.3h	2.0h	4.0h	12.0m	8.0m	1.4h	7.0m
PDF020	15.0s	15.0s	15.0s	15.0s	15.0s	15.0s	15.0s	15.0s
SQL001	×	×	×	×	×	×	×	×
SQL002	5.1h	23.0m	3.0m	28.0m	25.0s	1.0m	1.6h	4.0h
SQL003	×	×	10.9h	×	×	×	×	9.0h
SQL004	×	×	×	×	×	×	×	×
SQL005	×	×	×	×	×	×	×	18.2h
SQL006	×	×	1.0h	2.2h	×	×	4.0h	×
SQL007	51.0m	31.0m	3.0m	1.0h	20.0s	60.0s	59.1m	1.0h
SQL008	×	×	×	×	×	×	×	×
SQL009	×	8.7h	37.0m	18.0h	2.2h	38.0m	10.2h	13.0m
SQL010	12.6h	1.0h	12.0m	1.0h	25.0s	40.0m	2.0h	3.1h
SQL011	×	×	12.8h	×	×	×	13.5h	20.0h
SQL012	×	×	×	×	×	×	×	3.2h
SQL013	×	×	18.0h	×	3.0h	×	20.1h	×
SQL014	×	×	40.0m	20.9h	1.0h	×	2.3h	30.0m
SQL015	5.0h	23.0m	3.0m	1.1h	25.0s	1.0m	58.0m	1.0h
SQL016	6.5h	1.0h	5.0m	57.0m	25.0s	3.0m	31.0m	20.0s