



## Article

# MooFuzz: Many-Objective Optimization Seed Schedule for Fuzzer

Xiaoqi Zhao , Haipeng Qu <sup>\*</sup>, Wenjie Lv , Shuo Li  and Jianliang Xu

College of Information Science and Engineering, Ocean University of China, Qingdao 266100, China; zhaoxiaoqi@stu.ouc.edu.cn (X.Z.); lwj3656@stu.ouc.edu.cn (W.L.); li\_shuo@stu.ouc.edu.cn (S.L.); xjl9898@ouc.edu.cn (J.X.)

<sup>\*</sup> Correspondence: quhaipeng@ouc.edu.cn

**Abstract:** Coverage-based Greybox Fuzzing (CGF) is a practical and effective solution for finding bugs and vulnerabilities in software. A key challenge of CGF is how to select conducive seeds and allocate accurate energy. To address this problem, we propose a novel many-objective optimization solution, MooFuzz, which can identify different states of the seed pool and continuously gather different information about seeds to guide seed schedule and energy allocation. First, MooFuzz conducts risk marking in dangerous positions of the source code. Second, it can automatically update the collected information, including the path risk, the path frequency, and the mutation information. Next, MooFuzz classifies seed pool into three states and adopts different objectives to select seeds. Finally, we design an energy recovery mechanism to monitor energy usage in the fuzzing process and reduce energy consumption. We implement our fuzzing framework and evaluate it on seven real-world programs. The experimental results show that MooFuzz outperforms other state-of-the-art fuzzers, including AFL, AFLFast, FairFuzz, and PerFFuzz, in terms of path discovery and bug detection.



**Citation:** Zhao, X.; Qu, H.; Lv, W.; Li, S.; Xu, J. MooFuzz: Many-Objective Optimization Seed Schedule for Fuzzer. *Mathematics* **2021**, *9*, 205. <https://doi.org/10.3390/math9030205>

Academic Editors: Amir H. Alavi and Gai-Ge Wang

Received: 22 December 2020

Accepted: 16 January 2021

Published: 20 January 2021

**Publisher's Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



**Copyright:** © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

**Keywords:** seed schedule; many-objective optimization; fuzzing; bug detection; path discovery

## 1. Introduction

Fuzzing is a popular and effective software testing technology for detecting bugs and vulnerabilities. In the past few years, it has gained widespread usage in mainstream software companies (such as Google [1–3], Microsoft [4], and Adobe [5]) and has found thousands of vulnerabilities.

Coverage-based Greybox Fuzzing (CGF) [6,7] is one of the most popular methods of fuzzing. It is based on the guidance that increasing code coverage usually leads to better crash detection. By using lightweight instrumentation, CGF automatically generates a large number of inputs to feed target programs, and continuously collects coverage information as feedback to guide fuzzing.

Inspired by the impressive achievements of CGF, many researchers have conducted studies and developed their own fuzzers from different perspectives [8–10]. AFLFast [11] assigns more energy to the low-frequency paths based on the Markov chain model. AFLGo [12], a directed grey-box fuzzer, is implemented to generate inputs to reach given sets of target program locations. FairFuzz [13] identifies rare branches in the program and adjusts mutation strategies to increase coverage. MOPT [14] leverages a mutation schedule based on particle swarm optimization (PSO) to accelerate the convergence speed. EcoFuzz [15] improves the power schedule for discovering new paths using a variant of the adversarial multi-armed bandit model. PerFFuzz [16] generates pathological inputs to detect algorithm complexity vulnerabilities. MemLock [17] utilizes memory consumption information to guide seed selection to trigger the weakness of memory corruption.

However, most previous approaches mainly leverage a single selection criterion to select seeds. While these approaches are simple and easy to use in solving specific problems,

they are still inadequate to reach effective coverage and detect bugs within a reasonable amount of time. Cerebro [18] uses many objectives as the seed selection criteria, but it cannot dynamically adjust the seed selection strategy according to the fuzzing process. As a result, much useful information is ignored, affecting the discovery of bugs and paths.

In this paper, we propose a many-objective optimization seed schedule model, named MooFuzz, which is aimed at speeding up bug discovery and improving code coverage. MooFuzz performs static analysis on the code and marks the risky locations in order to collect edge risk information. In the fuzzing process, a novel measurement method is used to update useful information including the path risk, the path frequency, and the mutation information. According to the fuzzing process, MooFuzz divides the seed pool state into three categories: Exploration State, Search State, and Assessment State. In Exploration State, the fuzzer emphasizes the exploration of high-risk locations in the program. In Search State, the fuzzer spends more energy to find the new path. In Assessment State, the fuzzer aims to select and evaluate promising seeds. MooFuzz collects different information to measure the priority of seeds in each state and builds a many-objective optimization model to select optimal seed set using a non-dominated sorting algorithm [19]. Beside, we also observe that many studies have improved power schedule method, but they have not performed energy monitoring in power schedule. Therefore, MooFuzz uses multiple information to set the energy for selected seeds and monitors energy usage.

We design and implement our prototype by extending American Fuzzy Lop (AFL) [7], and evaluate it against for popular fuzzers AFL, AFLFast, FairFuzz, and PerFuzz in terms of path discovery and bug detection. We conduct our evaluation on seven real-world applications. The experimental results demonstrate that MooFuzz performs better than others. Compared with AFL, AFLFast, Fairfuzz, and PerFuzz, it triggers 46%, 32%, 34%, and 153% more crashes with almost the same execution time, respectively. Furthermore, we run cases and analyze the discovery of vulnerabilities. MooFuzz is able to trigger stack overflow, heap overflow, null pointer dereference, and memory leaks related vulnerabilities.

The contributions of this paper are as follows.

- We propose the path risk measurement method to assist seed schedule in Exploration State.
- We use many-objective optimization to model CGF and classify three different states of seed pool and put forth different selection criteria that enhance the fuzzer performance.
- We propose an energy allocation and monitor mechanism to improve the power schedule.
- We implement our framework as MooFuzz and evaluate its effectiveness on a series of popular real-world applications. MooFuzz substantially outperforms the other fuzzers.

The rest of this paper is organized as follows. Section 2 introduces the background and related work of many-objective optimization and CGF. Section 3 shows the design of MooFuzz. Section 4 presents the evaluation and we get the conclusion in Section 5.

## 2. Background and Related Work

In this section, we introduce the background of many-objective optimization and CGF and discuss related work.

### 2.1. Many-Objective Optimization

Multi-objective optimization [20–23] falls into the field of multiple criteria decision-making. It optimizes all goals at the same time to get the optimal solution. Therefore, multi-objective optimization problems (MOPs) get a set of solutions. Generally, a MOP is an optimization problem with two or three objectives. A many-objective optimization problem (MaOP) is an optimization problem [24–27] with four or more objectives. In recent years, many researchers have used multi-objective optimization methods to solve practical problems [28–31], such as scheduling [32,33], planning [34–36], fault diagnosis [37–39], classification [40,41], test-sheet composition [42], object extraction [43], variable reduction [44], and virtual machine placement [45]. Multi-objective evolutionary algorithms

(MOEAs), such as non-dominated sorting GA [46], multi-objective particle swarm optimization (MOPSO) [47–49], NSGA-II [50], NSGA-III [51,52], decomposition-based MOEA [53] and corresponding improved versions [54–56], are the most used solutions.

In many-objective optimization problems, minimization problems simultaneously optimize minimize objectives to obtain the maximum benefit. Within the scope of mathematics, minimization problems are embodied in the minimization of objective functions (that is, to minimize all objective values of objective functions as far as possible). In this paper, we use the minimum optimization model to carry out seed schedule. The definition of minimum optimization problems is given below.

$$\begin{cases} \text{Min } F(x) = [f_1(x), f_2(x), \dots, f_m(x)]^T \\ \text{s.t. } m > 3 \\ x \in X \subseteq \mathbb{R}^n \end{cases} \quad (1)$$

where  $F(x)$  is the objective vector,  $f_i(x)$  is the  $i$ -th objective to be minimized,  $x = (x_1, \dots, x_n)$  is a vector of  $n$  decision variables,  $X$  is an  $n$ -dimensional decision space, and  $m$  denotes the number of objectives to be optimized.

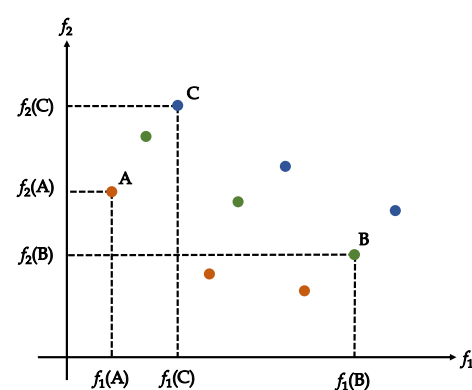
**Definition 1** (Pareto Dominance [57]). *Given any two decision vectors  $x, y$  with  $M$  objectives for the minimization optimization.  $\forall x, y \in X$ , if there is  $f_m(x) \leq f_m(y)$  for all  $m = 1, 2, \dots, M$  then  $x$  dominates  $y$ , which is denoted as  $x \prec y$ .*

**Definition 2** (Pareto Optimal [57]). *Assuming that  $x^* \in X$ , if there is no solution  $x \in X$  satisfying  $x \prec x^*$ , then  $x^*$  is the Pareto optimal solution.*

**Definition 3** (Pareto Optimal Set [57]). *All the Pareto optimal solutions constitute the Pareto optimal set (PS).*

**Definition 4** (Pareto Front [57]). *All the objective vectors of the solutions in Pareto optimal set constitute the Pareto front (PF).*

Figure 1 is a solution distribution under two-dimensional objective space, where all points represent solutions. For a minimal optimization problem, it can be seen that the point A is smaller than the point C under the two-dimensional objective space, that is, there is a dominance relationship between the point A and the point C, and the point C is dominated the point A. For the points A and B in Figure 1, we can see that the point A is greater than the point B on the  $f_2$  axis, but the point A is less than B on the  $f_1$  axis, so there is not a dominance relationship between the point A and the point B.



**Figure 1.** Solutions in a two-dimensional objective space.

## 2.2. Coverage-Based Greybox Fuzzing

CGF is an evolutionary algorithm that includes two stages: the static analysis stage and the fuzzing loop stage. In the static analysis stage, it executes compile-time or dynamic binary instrumentation to obtain the instrumented target program. In the fuzzing loop stage, CGF uses a series of initial seeds provided by the user as inputs and maintains a seed queue stored in the seed pool. CGF first selects a saved seed input from the seed queue and mutates it to generate the new input by using mutation strategies. Next, the target program is executed with the new input. Then, lightweight instrumentation technique is used to gather coverage information, if the new input causes a crash, it will be marked and added to the crash set. If the new input leads to new coverage, CGF will judge that the new input is interesting and add it to the seed pool. Algorithm 1 shows the workflow of CGF in the fuzzing loop stage.

---

### Algorithm 1: Coverage-based Greybox Fuzzing

---

**Input:** a data set of initial *seeds*, an instrumented target program *P*  
**Output:** a seed queue *Q*, a crash set *C*

```

1 Q ← seeds
2 C ← ∅
3 Procedure fuzzing process
4 while TRUE do
5   S ← SeedsSelect(Q)
6   E ← PowerSchedule(S)
7   for mutation in deterministic stage do
8     for i in Range(0, S.length) do
9       S' ← Mutation(S)
10      RunAndSave(P, S')
11    HavocMutation(P, E)
12    RunAndSave(P, S')
13 Procedure RunAndSave(P, S')
14 status ← Run_Target(P, S')
15 if is_NewCoverage(status) then
16   Q ← Q ∪ S'
17   return
18 if is_Crash(status) then
19   C ← C ∪ S'
20   return

```

---

### 2.2.1. Code Instrumentation

Code instrumentation aims to insert code fragments at compile-time, which is useful for path tracing and testing during the fuzzing process. AFL [7] is a greybox fuzzer using edge (branch) coverage as feedback. Before the fuzzing loop stage, AFL first uses afl-gcc or afl-clang as instrumentation commands to trace edge coverage. AFL preserves a 64KB shared bitmap *Bitmap* to record edge coverage information including whether the edge has been visited, and the count of hits. AFL assigns a random number to represent each basic block in the program and uses the XOR and right shift operation for the current basic block and the previous basic block to mark each edge. Each edge is used as an offset of *Bitmap* and the value is the count of hits.

The specific formula for coverage calculation is as follows [9].

$$cur\_location = Random() \quad (2)$$

$$Bitmap[cur\_location \oplus prev\_location] ++ \quad (3)$$

$$prev\_location = cur\_location >> 1 \quad (4)$$

### 2.2.2. Seed Schedule

Seed schedule refers to select seeds from the seed pool for future mutation. A perfect seed schedule scheme is conducive to speeding up path discovery and bug detection. AFL [7] gives priority to seeds that are unfuzzed (not selected for mutation) and favored (among all seeds passing through the edge, the seed with the smallest product of seed length and execution time). AFLGo [12] preferentially selects seeds closer to the target location for directed fuzzing. VUzzer [8] prioritizes seeds of deeper paths, it may detect bugs deep in the code. SlowFuzz [58] preferentially selects seeds that generate more resource consumption to trigger algorithm complexity vulnerabilities. In order to discover memory consumption bugs, MemLock [17] preferentially selects seed inputs that generate more memory consumption. UAFL [59] preferentially selects seeds that execute the operation sequence violating tpestate properties to uncover use-after-free (UAF) vulnerabilities.

### 2.2.3. Mutation Strategy

The mutation strategy determines where and how to mutate the selected seed. Different fuzzers use different mutation strategies. AFL has two mutation stages: the deterministic stage and the indeterministic stage.

**The deterministic stage.** The deterministic stage is used when the first time fuzzing seed. This stage includes mutation operators, bitflip, byteflip, arithmetic addition/subtraction, interesting values, and dictionary.

**The indeterministic stage.** After completing the deterministic stage, seeds will enter the indeterministic stage, in which AFL includes havoc and splice. In this stage, AFL randomly selects a sequences of mutation operators and assigns random location to mutate the seed.

There are many studies on mutation strategies for fuzzer. VUzzer [8] leverages data flow and control flow features to infer the critical regions of the input for mutation. GREYONE [60] uses a fuzzing-driven taint inference to infer taint variables for mutation. Superior [61] deploys mutation strategies to fuzz programs that process structured inputs. MOPT [14] uses particle swarm optimization algorithm to optimize mutation operators.

### 2.2.4. Power Schedule

Power schedule aims to allocate energy to each seed during the fuzzing process, which determines the number of seed mutations. Reasonable energy allocation can effectively improve the discovery of new paths. If the energy of a seed is over allocated, other seeds mutation will be affected. Conversely, if the energy of one seed is under allocated, it will be detrimental to new path discovery and potential bug detection.

AFL has two power schedule methods based on different mutation stages. In the deterministic stage, the energy of a seed is related to its length. The longer seed length, the more energy will be consumed. In the indeterministic stage, the energy allocation depends on the running time, the number of edges, the average size of the file, the number of cycles, and others.

Recent research shows that power schedule is very critical for fuzzer. AFLFast [11] allocates more energy to the low-frequency path to explore more paths. EcoFuzz [15] uses reinforcement learning to model power schedule as the adversarial multi-armed bandit model that enables adaptive energy saving. However, they did not consider the path risk and the effectiveness of energy allocation.

## 3. The Design of MooFuzz

To address problems mentioned in the previous sections, we propose a many-objective optimization fuzzer MooFuzz, as shown in Figure 2. The main components of MooFuzz contain static analyzer, feedback collector, seed scheduler, and power scheduler. In MooFuzz, static analyzer marks the risk edge and records the risk value for each edge by

scanning the source code and then inserts code fragments to update the edge risk value in running program. Feedback collector is used to record and update related information to guide the seed schedule after the program execution. Seed scheduler adopts different many-objective optimization schedules based on different states of the seed pool to select seeds. Power scheduler assigns energy based on feedback information and monitors energy usage.

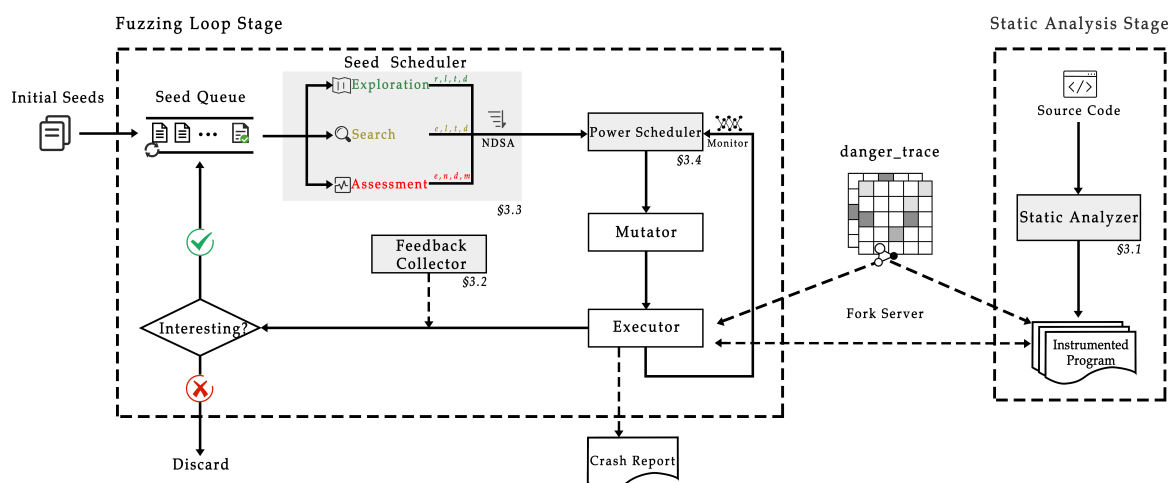


Figure 2. A high-level overview of MooFuzz.

### 3.1. Static Analyzer

A common idea is that the place has dangerous functions may trigger vulnerabilities. For example, the function *malloc* is used to dynamically allocate memory in C language. Although it can automatically allocate memory space, if used improperly, it may cause problems such as overflow, heap exhaustion, and use-after-free. The function *write* shall attempt to write *n* bytes from the buffer pointed to by *buf* into the file associated with the open file descriptor. However, if programmer cannot control the size of the bytes written to *buf*, it will cause the risk of out-of-bounds read of the memory. Therefore, MooFuzz identifies potentially dangerous functions as risk edges to label in static analyzer. In this paper, MooFuzz uses functions in Table 1 as dangerous functions [62], including memory allocation, memory recovery, memory operation, string operation, and file I/O operation. At the same time, users can also customize dangerous functions and add them to static analyzer for fuzzing.

Table 1. Dangerous functions.

Class	Description	Function Name
memory	memory allocation	<i>calloc; malloc; realloc</i>
	memory recovery	<i>free</i>
	memory operation	<i>memcpy; memmove; memchr; memset; memcmp</i>
string	string operation	<i>strcpy; strlen; strcat; strcmp; strspn; strcoll; strstr; strtok</i>
others	file I/O	<i>read; write</i>

Algorithm 2 shows the basic idea of MooFuzz instrumentation. Before the static analysis, there are well-known potentially dangerous functions. The static analyzer can identify them by traversing the source code and perform source code instrumentation at the corresponding edge position without running the program. MooFuzz uses a pointer *danger\_trace* to record the hit-counts of the risk edge in shared memory after running



program every time. Specifically, MooFuzz first obtains each basic block information of the program, then identifies each call instruction and judges whether someone is dangerous (Lines 1–7). If any exists, the hit-counts will be updated and stored in the memory pointed to by *danger\_trace* (Lines 8–11).

---

**Algorithm 2:** Code instrumentation
 

---

**Input:** the program  $P$ , a set of dangerous functions  $DF$ , a pointer variable *danger\_trace*

**Output:** the instrumented program  $P'$

```

1  $MAP\_SIZE = 2^{16}$ 
2 for basic_block in  $P$  do
3   bool risk = false
4   for CallInstr in basic_block do
5     if CallInstr.CalledFuncName in  $DF$  then
6       risk = true
7       break
8   cur = Random(0,  $MAP\_SIZE$ )
9   if risk=true then
10    danger_trace[cur  $\oplus$  pre] $++$ 
11    pre = cur  $>> 1$ 
  
```

---

### 3.2. Feedback Collector

The feedback collector is mainly used to continuously update seed information to assist seed schedule. For the running of the instrumented program, a series of running information would be updated for seeds. Algorithm 3 shows the process of information updating by feedback collector. It takes the seed queue  $Q$  and the pointer variable as inputs, and output is the seed queue  $Q'$  with new information. The new information includes the number of times the seed has been selected, the path frequency, the path risk, and the mutation information. Specifically, MooFuzz selects a seed  $s$  by using seed scheduler (see Section 3.3) and updates the number of times it has been selected (Lines 1–3). Then, it uses a mutation strategy to generate a new test case  $s'$  and executes the target program by using test case  $s'$  (Lines 4–5). Next, two pointer variables *danger\_bits* and *edge<sub>r</sub>* are used to update the edge risk (Line 6). Here, *danger\_bits* is obtained with the pointer variable *danger\_trace*. The *edge<sub>r</sub>* records the risk of each edge. At the beginning, the edge corresponding to dangerous function has a maximum value, while those of the other edges are zero. Next, if the mutated test case produces new coverage, MooFuzz will calculate path risk value (Lines 7–8). Next, MooFuzz traverses each seed in the seed pool and determine whether its path is the same as the current path. If so, the frequency information of the seeds in seed pools will be updated (Lines 9–11). Finally, if the path of  $s$  is identical to the path of  $s'$ , the mutation information will be updated (Lines 12–13).

We discuss how to update different information separately as follows.

The path risk mainly refers to the ability of seeds to detect dangerous locations, which determines the number and speed of bug discovery. Before discussing the path risk, we first give the definition of edge risk update and then that of path risk update.

**The edge risk update.** Given an edge  $e_i$  and the corresponding hit-count *danger\_bits*[ $e_i$ ], the edge risk *edge<sub>r</sub>*[ $e_i$ ] is updated as follows.

$$edge_r[e_i] = \begin{cases} edge_r[e_i] - danger\_bits[e_i], & e_i \in danger\_edge \\ 0, & others \end{cases} \quad (5)$$

where *danger\_edge* is the set of edges corresponding to dangerous function.

**Algorithm 3:** Information update

**Input:** a seed queue  $Q$ , a pointer variable  $danger\_bits$ , a pointer variable  $edge_r$ , the instrumented program  $P$

**Output:** a seed queue  $Q'$  with new information

```

1  $S = SeedSchedule(Q)$ 
2 for  $s$  in  $S$  do
3    $s.select\_num++$ 
4    $s' \leftarrow Mutation(s)$ 
5    $(trace\_bits, danger\_bits) \leftarrow Run\_target(s')$ 
6    $update\_edge\_risk(danger\_bits, edge_r)$ 
7   if  $is\_NewCoverage(P, s')$  then
8      $calculate\_path\_risk(edge_r)$ 
9   for  $s_i$  in seed pool do
10    if the path of  $s_i$  is the same as that of  $s'$  then
11       $update\_fre\_info(s_i)$ 
12  if the path of  $s'$  is the same as that of  $s$  then
13     $update\_mta\_info(s)$ 

```

**The path risk update.** Given a seed  $s$  and the risk values of all edges covered by the seed  $s$ , the path risk of seed  $s$ ,  $s.risk$  is calculated as follows.

$$s.risk = \sum_{i=1}^N \frac{edge_r[e_i]}{N} \quad (6)$$

The path frequency indicates the ability of the seed to discover a new path. As time goes by, there are high-frequency paths and low-frequency paths in the program. Generally, those seeds that cover low-frequency paths have a higher probability of discovering new paths than those that cover high-frequency paths (the larger the value, the higher the path frequency) after the program running for a while.

**The path frequency update.** Given a seed  $s'$  and its path  $p_{s'}$ , if there is a seed  $s$  in the seed pool and its path  $p_s$ , and  $p_s$  is the same as  $p_{s'}$ . We add one to the path frequency of seed  $s$ , that is,

$$s.fre = s.fre + 1, \text{ if } p_{s'} = p_s \quad (7)$$

The mutation information indicates the mutation ability of a seed. For each seed that has not been fuzzed, its mutation effectiveness is set to 0, indicating that the seed has the best mutation validity. Among the seeds being fuzzed, the mutation ability of the seeds will be continuously evaluated, and individuals with high mutation ability (the smaller the value, the better) will obtain priority.

**The mutation information update.** Given a seed  $s$  and its mutation strategy  $M$ , if the path of seed  $s$  is the same as that of seed  $s'$  generated by seed mutation upon  $s$ , the mutation information of seed  $s$ ,  $s.mta$  is calculated as follows.

$$s.mta = s.mta + 1, \text{ if } s' = M(s) \text{ and } p_{s'} = p_s \quad (8)$$

### 3.3. Seed Scheduler

Seed scheduler is mainly used for seeds selection. In order to effectively prioritize seeds, we propose a many-objective optimization seed schedule scheme.

Before seed schedule, MooFuzz divides the seed pool into three states according to seed attributes.



**Exploration State.** Exploration State refers to the existence of unfuzzed and favored seeds in the seed pool. Exploration State represents that the current seed pool state is an excellent state and it maintains the diversity of seeds.

**Search State.** In this state, the favored seeds have been fuzzed, but there are still unfuzzed seeds. Search State represents that there is a risk that the seed pool is completely fuzzed, and it is necessary to concentrate on finding more paths.

**Assessment State.** In this state, all the seeds are all fuzzed. It is very difficult to find a priority seed, but the fuzzed seeds produce a lot of information that can serve as a reference. Besides, MooFuzz performs state monitoring in the assessment state. Once the state changes, the seed set of the current state will be discarded to perform seed schedule in other states.

For these three states, MooFuzz uses different selection criteria based on bug detection, path discovery, and seed evaluation. MooFuzz constructs different objective functions based on different states.

In the previous discussion, MooFuzz has obtained the risk value of the seed before it is added to the seed pool, indicating the path risk. Based on previous research [8], seeds with deeper executing paths may be more capable of detecting bugs. Therefore, MooFuzz uses path risk  $r$  and path depth  $d$  as objectives for seed selection. To reduce the energy consumption of seeds and speed up the discovery of bugs, MooFuzz also takes the length  $l$  of the seed data and the execution time  $t$  of the seed as objectives. In Exploration State, MooFuzz uses the following objective functions to select the seeds that have not been fuzzed and favored.

$$\text{Min } F(s) = [-r, -d, l, t]^T, s \in S \quad (9)$$

Search State indicates that all the favored seeds in current seed pool have been fuzzed and there are unfuzzed seeds. At this time, MooFuzz's selection of seeds will mainly focus on the path discovery. The frequency information of the seeds will increase with the running time changes. In this state, those seeds that pass the low-frequency path will have greater potential to discover new paths. MooFuzz regards path frequency  $e$  and path depth  $d$  as criteria for seeds selection. Meanwhile, MooFuzz uses  $l$  and  $t$  described above to balance energy consumption. In Search State, MooFuzz uses the following objective functions to select the seeds that have not been fuzzed.

$$\text{Min } F(s) = [e, -d, l, t]^T, s \in S \quad (10)$$

Assessment State means that all seeds in the current seed pool have been fuzzed. MooFuzz will obtain the information of the seed including the path frequency  $e$ , the number of times that the seed has been selected  $n$ , the seed path depth  $d$ , and the mutation information  $m$ , and then add them to the objective functions as mutation criterion. Note that the current state does not choose the length and execution time of the seed as criteria to balance energy consumption, because the current state is very difficult to generate new seeds. Besides, once new seeds are generated in this state, Assessment State will be terminated and enter other state. In Assessment State, MooFuzz uses the following objective functions to select the seeds from the seed pool.

$$\text{Min } F(s) = [e, n, -d, m]^T, s \in S \quad (11)$$

MooFuzz selects the optimal seed set after establishing objective functions for different seed pool states and models seed schedule as a minimization problem. Algorithm 4 mainly completes the seed schedule by using non-dominated sorting [19]. The seed set  $S$  that satisfies state conditions will be selected as the input. A set  $CF$  that is used to store the optimal seed set. Initially,  $CF$  is an empty set, and  $s_1$  in seed set  $S$  was added to  $CF$ . For each seed  $s_i$  from the seed set  $S$  and seeds  $s_j$  in  $CF$  finish the dominance comparisons (Lines 1–9). If  $s_j$  dominates  $s_i$  (each attribute value of  $s_j$  is less than  $s_i$ ), the next seed comparison will be performed. If  $s_i$  dominates  $s_j$ , remove  $s_j$  from  $CF$ . After the comparison between the seed  $s_i$  and  $s_j$ , if there is not a dominance relationship between  $s_i$  and all the seeds in

$CF$ ,  $s_i$  will be added to  $CF$  (Lines 10–11). After the above cycle is completed, the optimal seed set is stored in  $CF$ , and MooFuzz extracts each seed inside for fuzzing (Lines 12–13).

---

**Algorithm 4:** Seed schedule
 

---

**Input:** the seed set  $S$  satisfying conditions in different states  
**Output:** a series of optional seed  $s'$

```

1  $CF \leftarrow \emptyset$ 
2 for  $s_i$  in  $S$  do
3   bool  $isdominated = false$ 
4   for  $s_j$  in  $CF$  do
5     if  $s_j$  dominates  $s_i$  then
6        $isdominated = true$ 
7       break
8     if  $s_i$  dominates  $s_j$  then
9        $CF \leftarrow CF - s_j$ 
10  if not  $isdominated$  then
11     $CF \leftarrow CF \cup s_i$ 
12 for  $s'$  in  $CF$  do
13    $fuzz(s')$ 
  
```

---

### 3.4. Power Scheduler

The purpose of power schedule is assigning reasonable energy for each seed involved in mutation. A high quality seed has more chances to mutation and should be assigned with more energy in fuzzing process.

Existing coverage-based fuzzers (such as AFL [7]) usually calculate the energy for the selected seeds as follows [18],

$$energy(i) = allocate\_energy(q_i) \quad (12)$$

where  $i$  is the seed and  $q_i$  is the quality of the seed, depending on the execution time, branch edge coverage, creation time, and so on.

Algorithm 5 is the seed power schedule algorithm. MooFuzz considers different seed pool states to set up different energy distribution methods. Meanwhile, it also uses an energy monitoring mechanism, which has the ability to monitor the execution of target programs and reduce unnecessary energy consumption.

After many experiments, we find that the amount of energy in the deterministic stage is mainly related to the length of the seed, which is a relatively fine-grained mutation, but as the number of candidate seeds in the seed pool increases, it will affect the path discovery. Thus, in Algorithm 5 we open the deterministic stage to seeds that cause crashes after mutation (Lines 1–2). In the indeterministic stage, MooFuzz judges the state of the current seed. If it belongs to Search State, MooFuzz uses the frequency information to set the energy. If it belongs to Assessment State, both the frequency and the mutation information will be comprehensively considered to set the energy (Lines 3–6).

After energy allocation, we set up a monitoring mechanism to monitor the mutation of seeds (Lines 7–14). When each seed consumes 75% of the allocated energy, MooFuzz monitors the mutation of the current seed, and records the ratio of the average energy consumption of the current seed covering a new path and that of all seeds covering a new path. If its ratio is lower than  $threshold_1$ , MooFuzz will withdraw the energy, if its ratio is higher than  $threshold_2$ , the mutation information will be updated. Here,  $threshold_1$  is equal to 0.9 and  $threshold_2$  is equal to 1.3.

**Algorithm 5:** Power schedule

---

**Input:** a seed  $s$ , the number of all seeds in seed pool  $total\_seed$ , the total energy consumed in the fuzzing process  $total\_energy$ , the number of new seeds generated by the current seed mutation  $cur\_seed$

**Output:** the energy of seed  $s$   $s.energy$

```

1 if seed  $s$  that causes crashes after mutation then
2   goto deterministic stage
  /* indeterministic stage: */
3 if state is Search State then
4    $s.energy = (1 + \frac{1}{s.fre}) * energy(s)$ 
5 if state is Assessment State then
6    $s.energy = (1 + (\frac{1}{s.mta} + \frac{1}{s.fre})) * energy(s)$ 
7 for  $cur\_energy = 0$  to  $s.energy$  do
8   if the energy consumption of seed  $s$  reaches 75% then
9      $total\_average = \frac{total\_energy}{total\_seed}$ 
10     $cur\_average = \frac{cur\_energy}{cur\_seed}$ 
11    if  $\frac{cur\_average}{total\_average} < threshold_1$  then
12      break
13    if  $\frac{cur\_average}{total\_average} > threshold_2$  then
14       $s.mta = s.mta * 0.9$ 

```

---

**4. Evaluation**

MooFuzz is built on top of AFL-2.52b [7]. The implementation adds C/C++ code to the AFL. The instrumentation components are implemented to mark danger edges based on the LLVM framework [63] in static analysis. Through these experiments, the following research questions are tackled:

- RQ1: How capable is MooFuzz in crash detection?
- RQ2: How effective is the code coverage of MooFuzz?
- RQ3: How capable is MooFuzz in identifying real-world vulnerabilities?

**4.1. Experimental Settings**

**Baseline Fuzzer.** We compare MooFuzz with existing state-of-the-art tools AFL [7], AFLFast [11], FairFuzz [13], and PerfFuzz [16]. The selection of baseline fuzzer is mainly based on the following considerations:

1. AFL is currently one of the most common coverage-based greybox fuzzer in community.
2. AFLFast is a variant of AFL with better power schedule.
3. FairFuzz is also an extending fuzzer of AFL. It optimizes inputs that hit rare branches.
4. PerfFuzz improves the instrumented components to generate pathological inputs.

**Benchmark.** To evaluate MooFuzz, we choose seven real-world open source Linux applications as the benchmark to conduct experiments. Jasper [64] is a software tool kit for processing image data that provides a way to represent images and facilitates the manipulation of image data. LibSass [65] is a C/C++ port of the Sass engine. Exiv2 [66] is a C++ library and a command line utility to read, write, delete, and modify Exif, IPTC, XMP, and ICC image metadata. Libming [67] is a library for generating Macromedia Flash files, written in C, and includes useful utilities for working with Flash files. OpenJPEG [68] is an open source JPEG 2000 codec written in C language. Bento4 [69] is a C++ class library that

is designed to read and write ISO-MP4 files. The GUN Binutils [70] is a collection of binary tools. Table 2 shows target applications and their fuzzing configure.

**Table 2.** Target applications and their fuzzing configure.

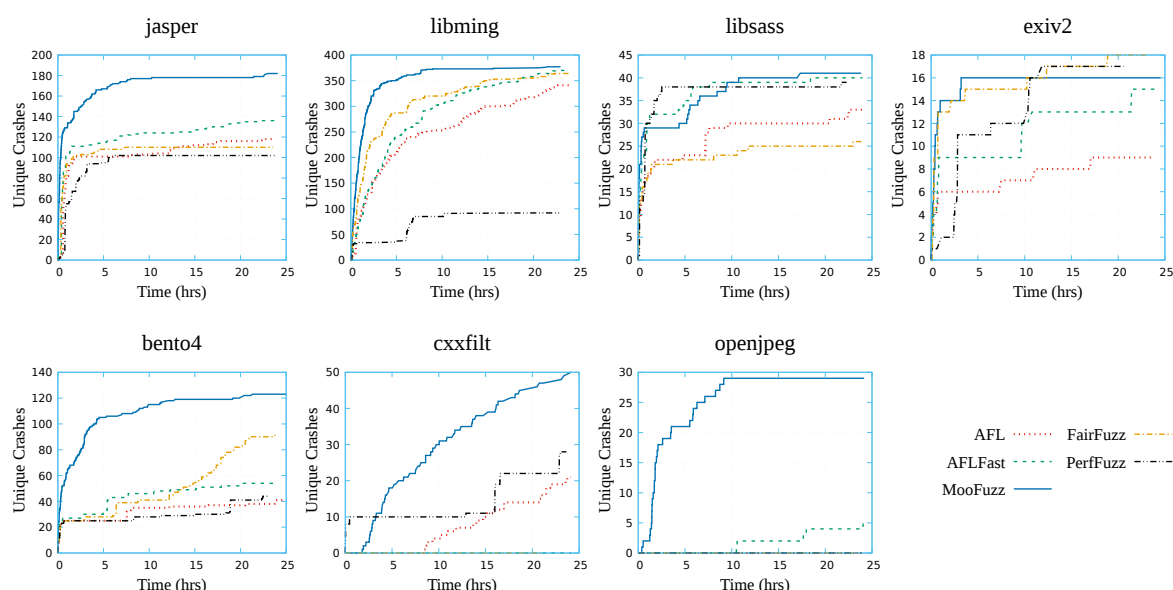
Program	Command Line	Project Version
jasper	jasper --input @@ --output t.bmp	Jasper 2.0.14
libsass	tester @@	LibSass 3.5.4
exiv2	exiv2 -pX @@	Exiv2 0.26
libming	listswf @@	Libming 0.4.8
openjpeg	opj_decompress -i @@ -o t.png	OpenJPEG 0.26
cxxfilt	c++filt -t	GUN Binutils 2.31
bento4	mp42hls @@	Bento4 1.5.1

**Performance Metrics.** Crashes, paths, and vulnerabilities are chosen as metrics in this section. In code coverage metrics, we use the number of seeds in the queue as an indicator and use tool Afl-cov [71] to measure code line coverage and function coverage. In vulnerability detection, we directly use AddressSanitizer [72] to detect it.

**Experiment Environment.** All experiments are conducted on a server configured with two Xeon E5-2680 v4 processors (56 logical cores in total) and 32 GB RAM. The server installed Ubuntu 18.04 system. For the same application, the initial seed set is the same. We fuzz each application for 24 h (on a single logical core) and repeat 5 times to reduce randomness. In all implementations, we use 42 logical cores, and we leave 14 logical cores for other processes to keep the workload stable.

#### 4.2. Unique Crashes Evaluation (RQ1)

In order to evaluate the effectiveness of MooFuzz, a direct method is to evaluate the number of crashes and the speed at which they are triggered. It is believed that more crashes may trigger more bugs. We fuzz each application to run on 5 different fuzzers to compare the number of unique crashes and the speed of discovery. Figure 3 shows the growth trends of unique crashes discovery in different fuzzers. From these results, we can make the follow observations.



**Figure 3.** The growth trends of unique crashes discovery in different fuzzers within 24 h.

First, different fuzzers have different capability in fuzzing different application programs. For example, PerfFuzz has zero crash in fuzzing openjpeg within 24 h, but it can

trigger most crashes in fuzzing exiv2 among other fuzzers. This shows that the different criteria of the seed selection affect the number of crashes.

Second, seed schedule and power schedule affect the efficiency of crashes discovery. The experimental results show that MooFuzz outperforms AFL in the speed of crashes discovery and just takes about 10 h to trigger most of the unique crashes. There is no path risk measurement and energy monitoring in AFL, leading to a lot of time spent on invalid mutation operators.

Third, MooFuzz is able to find more crashes than other state-of-the-art fuzzers. The static results are shown in Table 3. We count the number of crashes found in applications by different fuzzers within 24 h, and count the total number of crashes found by each fuzzer. Table 3 shows that except for exiv2, MooFuzz triggers more crashes than other fuzzers, among which jasper triggers 182 crashes within 24 h and AFL only triggers 118 crashes. In total, MooFuzz triggers 818 crashes in benchmark application programs, improving by 46%, 32%, 34%, and 153%, respectively, compared with state-of-the-art fuzzers AFL [7], AFLFast [11], FairFuzz [13], and PerfFuzz [16].

**Table 3.** Number of unique crashes found in real-world programs by various fuzzers.

Program	MooFuzz	AFL	AFLFast	FairFuzz	PerfFuzz
jasper	182	118	136	110	102
libming	377	341	371	364	92
libsass	41	33	40	26	39
exiv2	16	9	15	18	17
bento4	123	40	54	91	44
cxxfilt	50	21	0	0	29
openjpeg	29	0	5	0	0
total	818	562	621	609	323

Overall, MooFuzz significantly outperforms other fuzzers in terms of speed and number of unique crashes.

#### 4.3. Coverage Evaluation (RQ2)

Code coverage is an effective way to evaluate fuzzers. The experiment measures coverage from source code line, function, and path. Table 4 shows the line and function covered by different fuzzers. In total, MooFuzz's line coverage and function coverage are better than AFL, AFLFast, FairFuzz, and PerfFuzz.

**Table 4.** Line and function covered by fuzzers.

Program	MooFuzz		AFL		AFLFast		FairFuzz		PerfFuzz	
	Line	Func	Line	Func	Line	Func	Line	Func	Line	Func
jasper	32.8%	47.5%	32.1%	46.4%	32.2%	46.7%	31.4%	45.8%	32.2%	46.7%
libming	15.5%	16.8%	13.6%	14.8%	13.0%	14.3%	16.1%	17.3%	6.0%	7.4%
libsass	52.2%	35.0%	51.1%	35.1%	50.2%	34.5%	52.2%	35.3%	45.3%	32.8%
exiv2	5.0%	9.0%	4.9%	8.8%	4.9%	8.9%	4.9%	8.8%	4.9%	8.8%
bento4	12.1%	12.6%	11.4%	11.5%	11.4%	11.5%	11.5%	11.7%	11.5%	11.7%
cxxfilt	2.5%	3.0%	2.5%	3.0%	2.7%	3.1%	2.8%	3.1%	1.5%	2.5%
openjpeg	31.2%	41.4%	29.2%	33.5%	31.7%	41.3%	33.2%	41.9%	29.5%	39.0%

Figure 4 shows the growth trends of paths discovery in five different fuzzers after fuzzing applications for 24 h. We can clearly observe that except for cxxfilt, MooFuzz ranks first among all fuzzers from the perspective of the number of path discovery. Among them, it can find about 6000 paths in fuzzing openjpeg, and the other four fuzzers can only find about 3600 paths. It can find about 1200 paths after fuzzing jasper for 24 h, while other fuzzers can only find about 500 to 700 paths. Although the number of paths discovered

by MooFuzz is lower than FairFuzz and AFLFast in fuzzing cxxfilt, it can trigger the most crashes compared with other fuzzers. From the speed of path discovery, MooFuzz is significantly higher than other fuzzers.

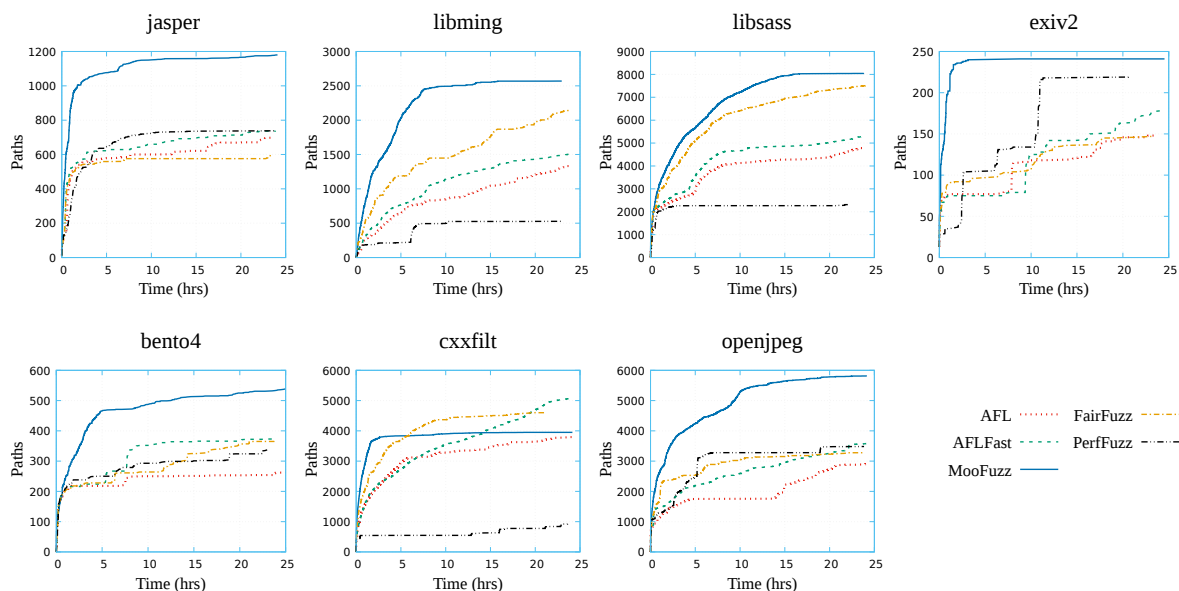


Figure 4. The growth trends of paths discovery in different fuzzers within 24 h.

Overall, MooFuzz outperforms other fuzzers in terms of line, function, and path coverage.

#### 4.4. Vulnerability Evaluation (RQ3)

MooFuzz tests old version of the applications and analyzes related vulnerabilities to evaluate the ability in vulnerability detection. Table 5 shows the real vulnerabilities combination with its IDs identified by MooFuzz. MooFuzz is able to find stack overflow, heap overflow, null pointer dereference, and memory leaks related vulnerabilities.

Table 5. Real-world vulnerabilities found by MooFuzz.

Program	CVE	Vulnerability
cxxfilt	CVE-2018-9138	stack overflow
cxxfilt	CVE-2018-17985	stack overflow
jasper	CVE-2018-19543	out-of-bounds read
jasper	CVE-2018-19542	null pointer dereference
jasper	CVE-2018-19541	out-of-bounds read
libsass	CVE-2018-19837	stack overflow
libsass	CVE-2018-20821	stack overflow
libsass	CVE-2018-20822	stack overflow
exiv2	CVE-2018-16336	heap-buffer-overflow
exiv2	CVE-2018-17229	heap-buffer-overflow
exiv2	CVE-2018-17230	heap-buffer-overflow
exiv2	CVE-2017-14861	stack-buffer-overflow
libming	CVE-2018-13066	memory leaks
libming	CVE-2020-6628	heap-buffer-overflow
openjpeg	CVE-2020-8112	heap-buffer-overflow
bento4	CVE-2019-15050	out-of-bounds read
bento4	CVE-2019-15048	heap-buffer-overflow

**Vulnerability analysis.** We use a real-world application program vulnerability to analyze the effectiveness of our approach, as shown in Figure 5. This is a code snippet from openjpeg [68] which contains a heap-buffer-overflow vulnerability (i.e., CVE-2020-8112).



In Figure 5, the main function contains a conditional statement (Lines 1–9). In MooFuzz, the seed  $s_t$  satisfies the judgment condition and enters the true branch to execute function *opj\_tcd\_decode\_tile(...)*. Moreover, the seed  $s_n$  enters the false branch to execute other codes that do not contain dangerous functions. *Asmalloc* is a dangerous function which is used in *opj\_tcd\_decode\_tile(...)*, risks might emerge when this function is used. Therefore, MooFuzz preferentially selects seed  $s_t$  for mutation. In this case, *malloc(l\_data\_size)* is called and *l\_data\_size* comes from an unsigned operation in the function *opj\_tcd\_decode\_tile*. Then, the function *opj\_t1\_clbl\_decode\_processor* will be called in the following program flow, where the allocated memory will be modify through two variables *cblk\_h* and *cblk\_w*. All of these two variables are obtained through signed operation, which causes an integer overflow making *cblk\_h \* cblk\_w > l\_data\_size*, and MooFuzz easily satisfies the above conditions through mutation, so the heap-buffer-overflow happened.

```

1  int main(int argc, char **argv){
2  ...
3  if (!parameters.nb_tile_to_decode){
4  opj_tcd_decode_tile(...)
5  }
6  else{
7  ...
8  }
9  }
10
11 OPJ_BOOL opj_tcd_decode_tile(opj_tcd_t *p_tcd, ...){
12 /* unsigned operations */
13 OPJ_SIZE_T res_w = (OPJ_SIZE_T)(l_res->x1 - l_res->x0);
14 OPJ_SIZE_T res_h = (OPJ_SIZE_T)(l_res->y1 - l_res->y0);
15
16 l_data_size = res_w * res_h;
17 /* tile->data allocate l_data_size space */
18 tilec->data = malloc(l_data_size);
19 ...
20 opj_t1_clbl_decode_processor(...);
21 }
22
23 static void opj_t1_clbl_decode_processor(...){
24 /* cblk_h and cblk_w are obtained through signed operation
25 * which cause integer overflow
26 * cblk_h * cblk_w > l_data_size, heap overflow happened. */
27 datap = tilec->data;
28 for (j = 0; j < cblk_h; ++j){
29 i = 0;
30 for (; i < (cblk_w & ~(OPJ_UINT32)3U); i += 4U){
31 OPJ_INT32 tmp0 = datap[(j * cblk_w) + i + 0U];
32 ...
33 }
34 }
35 }

```

Figure 5. An example from openjpeg (CVE-2020-8112).

#### 4.5. Discussion

We enhance fuzzing from the perspectives of vulnerabilities and coverage. Although more coverage may trigger more vulnerabilities, not all coverage is equal [62]. Based on our observation of the fuzzing process, we define the path risk and prioritize seeds that consume less energy while executing high risks, to maximize the improvement of fuzzing. Meanwhile, we use different objectives for seed optimization and energy allocation. It can improve the efficiency of fuzzing in a limited time.

In the algorithm design of the power schedule, we use two thresholds to judge the current seed energy usage. There is still an opportunity to adaptively adjust these two thresholds instead of the fixed thresholds. For example, these thresholds can be dynamically adjusted according to the fuzzing process. In our evaluation, our method can improve the probability of triggering vulnerabilities, but it may not be effective for triggering vulnerabilities that require complex conditions, such as deeply nested conditions. Although we use a variety of open source benchmarks to evaluation MooFuzz, it may not be effective for programs that require specific grammatical conditions for inputs (such as XML). However, the prototype we develop, MooFuzz, is a completely dynamic prototype.



It can integrate static analysis techniques like symbolic execution to generate test cases that satisfy specific conditions to improve fuzzing.

## 5. Conclusions and Further Work

In this paper, a many-objective optimization model is built for seed schedule. Considering the three states of the seed pool, we use different objective functions to select seeds from the perspectives of bug detection, path discovery, and seed evaluation. At the same time, an energy recovery mechanism is designed to monitor energy usage during the fuzzing process. We implement a prototype MooFuzz on top of AFL and evaluate it on seven real-world programs. The experiment results show that MooFuzz behaves more effectively than state-of-the-art fuzzers in path discovery and bug detection.

In the future, we plan to use MooFuzz to fuzz the latest version of the applications to assist testers in testing. In next study, we will consider optimizing power schedule through multi-information feedback on the basis of MooFuzz, so that it can monitor energy consumption according to the current program running progress, and automatically set and adjust energy. We also consider starting from the seed mutation, and propose a new decision model to determine effective region of the seed and select the effective mutation strategy.

**Author Contributions:** Conceptualization, X.Z. and H.Q.; methodology, X.Z. and H.Q.; software, W.L. and X.Z.; validation, W.L., S.L., and X.Z.; formal analysis, H.Q.; investigation, X.Z. and S.L.; resources, J.X.; data curation, W.L.; writing—original draft preparation, X.Z.; writing—review and editing, X.Z. and J.X.; visualization, W.L. and S.L.; supervision, H.Q.; project administration, X.Z.; funding acquisition, H.Q. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research was funded by the National Natural Science Foundation of China grant number 61827810.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** The test results data presented in this study are available on request. The data set can be found in public web sites.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Arya, A.; Neekar, C. Fuzzing for Security. Available online: <https://blog.chromium.org/2012/04/fuzzing-for-security.html> (accessed on 30 November 2020).
2. Evans, C.; Moore, M.; Ormandy, T. Fuzzing at Scale. Available online: <https://security.googleblog.com/2011/08/fuzzing-at-scale.html> (accessed on 30 November 2020).
3. Moroz, M.; Serebryany, K. Guided in-Process Fuzzing of Chrome Components. Available online: <https://security.googleblog.com/2016/08/guided-in-process-fuzzing-of-chrome.html> (accessed on 30 November 2020).
4. Godefroid, P.; Kiezun, A.; Levin, M.Y. Grammar-based whitebox fuzzing. In Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008), Tucson, AZ, USA, 21–25 June 2008; pp. 206–215. [CrossRef]
5. Arkin, B. Adobe Reader and Acrobat Security Initiative. Available online: [https://blogs.adobe.com/security/2009/05/adobe\\_reader\\_and\\_acrobat\\_sec.html](https://blogs.adobe.com/security/2009/05/adobe_reader_and_acrobat_sec.html) (accessed on 30 November 2020).
6. Serebryany, K. Continuous fuzzing with libFuzzer and AddressSanitizer. In Proceedings of the 2016 IEEE Cybersecurity Development (SecDev 2016), Boston, MA, USA, 3–4 November 2016; p. 157. [CrossRef]
7. Zlewski, C. American Fuzzy Lop. Available online: <http://lcamtuf.coredump.cx/afl> (accessed on 1 September 2020).
8. Rawat, S.; Jain, V.; Kumar, A.; Cojocar, L.; Giuffrida, C.; Bos, H. VUzzer: Application-aware evolutionary fuzzing. In Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS 2017), San Diego, CA, USA, 26 February–1 March 2017; pp. 1–14. [CrossRef]
9. Gan, S.; Zhang, C.; Qin, X.; Tu, X.; Li, K.; Pei, Z.; Chen, Z. Collafl: Path sensitive fuzzing. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (S&P 2018), San Francisco, CA, USA, 21–23 May 2018; pp. 679–696. [CrossRef]
10. Sun, L.; Li, X.; Qu, H.; Zhang, X. AFLTurbo: Speed up path discovery for greybox fuzzing. In Proceedings of the 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE 2020), Coimbra, Portugal, 12–15 October 2020; pp. 81–91. [CrossRef]

11. Böhme, M.; Pham, V.; Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. *IEEE Trans. Softw. Eng.* **2019**, *45*, 489–506. [\[CrossRef\]](#)
12. Böhme, M.; Pham, V.T.; Nguyen, M.D.; Roychoudhury, A. Directed greybox fuzzing. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017), Dallas, TX, USA, 30 October–3 November 2017; pp. 2329–2344. [\[CrossRef\]](#)
13. Lemieux, C.; Sen, K. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018), Montpellier, France, 3–7 September 2018; pp. 475–485. [\[CrossRef\]](#)
14. Lyu, C.; Ji, S.; Zhang, C.; Li, Y.; Lee, W.H.; Song, Y.; Beyah, R. MOPT: Optimized mutation scheduling for fuzzers. In Proceedings of the 28th USENIX Security Symposium (USENIX Security 2019), Santa Clara, CA, USA, 14–16 August 2019; pp. 1949–1966.
15. Yue, T.; Wang, P.; Tang, Y.; Wang, E.; Yu, B.; Lu, K.; Zhou, X. EcoFuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 2020), Vancouver, BC, Canada, 12–14 August 2020; pp. 2307–2324.
16. Lemieux, C.; Padhye, R.; Sen, K.; Song, D. PerFuzz: Automatically generating pathological inputs. In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018), Amsterdam, The Netherlands, 16–21 July 2018; pp. 254–265. [\[CrossRef\]](#)
17. Wen, C.; Wang, H.; Li, Y.; Qin, S.; Liu, Y.; Xu, Z.; Chen, H.; Xie, X.; Pu, G.; Liu, T. Memlock: Memory usage guided fuzzing. In Proceedings of the 42nd International Conference on Software Engineering (ICSE 2020), Han River, Seoul, Korea, 6–11 July 2020; pp. 765–777. [\[CrossRef\]](#)
18. Li, Y.; Xue, Y.; Chen, H.; Wu, X.; Zhang, C.; Xie, X.; Wang, H.; Liu, Y. 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 (FSE 2019), Tallinn, Estonia, 26–30 August 2019; pp. 533–544. [\[CrossRef\]](#)
19. Yuan, Y.; Xu, H.; Wang, B. An improved NSGA-III procedure for evolutionary many-objective optimization. In Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation (GECCO 2014), Vancouver, BC, Canada, 12–16 July 2014; pp. 661–668. [\[CrossRef\]](#)
20. Rizk-Allah, R.M.; El-Sehiemy, R.A.; Deb, S.; Wang, G.G. A novel fruit fly framework for multi-objective shape design of tubular linear synchronous motor. *J. Supercomput.* **2017**, *73*, 1235–1256. [\[CrossRef\]](#)
21. Li, J.; Lei, H.; Alavi, A.H.; Wang, G.G. Elephant herding optimization: Variants, hybrids, and applications. *Mathematics* **2020**, *8*, 1415. [\[CrossRef\]](#)
22. Sun, J.; Miao, Z.; Gong, D.; Zeng, X.J.; Li, J.; Wang, G. Interval multiobjective optimization with memetic algorithms. *IEEE Trans. Cybern.* **2020**, *50*, 3444–3457. [\[CrossRef\]](#)
23. Wang, F.; Li, Y.; Liao, F.; Yan, H. An ensemble learning based prediction strategy for dynamic multi-objective optimization. *Appl. Soft Comput.* **2020**, *96*, 106592. [\[CrossRef\]](#)
24. Feng, Y.; Wang, G.G.; Li, W.; Li, N. Multi-strategy monarch butterfly optimization algorithm for discounted {0-1} knapsack problem. *Neural Comput. Appl.* **2018**, *30*, 3019–3036. [\[CrossRef\]](#)
25. Srikanth, K.; Panwar, L.K.; Panigrahi, B.K.; Herrera-Viedma, E.; Sangaiah, A.K.; Wang, G.G. Meta-heuristic framework: Quantum inspired binary grey wolf optimizer for unit commitment problem. *Comput. Electr. Eng.* **2018**, *70*, 243–260. [\[CrossRef\]](#)
26. Feng, Y.; Deb, S.; Wang, G.G.; Alavi, A.H. Monarch butterfly optimization: A comprehensive review. *Expert Syst. Appl.* **2021**, *168*, 114418. [\[CrossRef\]](#)
27. Pan, Q.K.; Sang, H.Y.; Duan, J.H.; Gao, L. An improved fruit fly optimization algorithm for continuous function optimization problems. *Knowl.-Based Syst.* **2014**, *62*, 69–83. [\[CrossRef\]](#)
28. Sang, H.Y.; Pan, Q.K.; Duan, P.Y. Self-adaptive fruit fly optimizer for global optimization. *Nat. Comput.* **2019**, *18*, 785–813. [\[CrossRef\]](#)
29. Wang, F.; Li, Y.; Zhou, A.; Tang, K. An estimation of distribution algorithm for mixed-variable newsvendor problems. *IEEE Trans. Evol. Comput.* **2020**, *24*, 479–493. [\[CrossRef\]](#)
30. Wang, G.G.; Guo, L.; Gandomi, A.H.; Hao, G.S.; Wang, H. Chaotic krill herd algorithm. *Inf. Sci.* **2014**, *274*, 17–34. [\[CrossRef\]](#)
31. Wang, G.G.; Deb, S.; Cui, Z. Monarch butterfly optimization. *Neural Comput. Appl.* **2019**, *31*, 1995–2014. [\[CrossRef\]](#)
32. Gao, D.; Wang, G.G.; Pedrycz, W. Solving fuzzy job-shop scheduling problem using DE algorithm improved by a selection mechanism. *IEEE Trans. Fuzzy Syst.* **2020**, *28*, 3265–3275. [\[CrossRef\]](#)
33. Sang, H.Y.; Pan, Q.K.; Duan, P.Y.; Li, J.Q. An effective discrete invasive weed optimization algorithm for lot-streaming flowshop scheduling problems. *J. Intell. Manuf.* **2018**, *29*, 1337–1349. [\[CrossRef\]](#)
34. Wu, G.; Pedrycz, W.; Li, H.; Ma, M.; Liu, J. Coordinated planning of heterogeneous earth observation resources. *IEEE Trans. Syst. Man, Cybern. Syst.* **2015**, *46*, 109–125. [\[CrossRef\]](#)
35. Wang, G.G.; Gandomi, A.H.; Yang, X.S.; Alavi, A.H. A new hybrid method based on krill herd and cuckoo search for global optimisation tasks. *Int. J. Bio-Inspired Comput.* **2016**, *8*, 286–299. [\[CrossRef\]](#)
36. Wang, G.G.; Guo, L.; Duan, H.; Liu, L.; Wang, H.; Shao, M. Path planning for uninhabited combat aerial vehicle using hybrid meta-heuristic DE/BBO algorithm. *Adv. Sci. Eng. Med.* **2012**, *4*, 550–564. [\[CrossRef\]](#)
37. Yi, J.H.; Wang, J.; Wang, G.G. Improved probabilistic neural networks with self-adaptive strategies for transformer fault diagnosis problem. *Adv. Mech. Eng.* **2016**, *8*, 1–13. [\[CrossRef\]](#)

38. Mao, W.; He, J.; Li, Y.; Yan, Y. Bearing fault diagnosis with auto-encoder extreme learning machine: A comparative study. *Proc. Inst. Mech. Eng. Part C J. Mech. Eng. Sci.* **2017**, *231*, 1560–1578. [\[CrossRef\]](#)
39. Mao, W.; Feng, W.; Liang, X. A novel deep output kernel learning method for bearing fault structural diagnosis. *Mech. Syst. Signal Process.* **2019**, *117*, 293–318. [\[CrossRef\]](#)
40. Wang, G.G.; Lu, M.; Dong, Y.Q.; Zhao, X.J. Self-adaptive extreme learning machine. *Neural Comput. Appl.* **2016**, *27*, 291–303. [\[CrossRef\]](#)
41. Mao, W.; Zheng, Y.; Mu, X.; Zhao, J. Uncertainty evaluation and model selection of extreme learning machine based on Riemannian metric. *Neural Comput. Appl.* **2014**, *24*, 1613–1625. [\[CrossRef\]](#)
42. Duan, H.; Zhao, W.; Wang, G.G.; Feng, X.H. Test-sheet composition using analytic hierarchy process and hybrid metaheuristic algorithm TS/BBO. *Math. Probl. Eng.* **2012**, *2012*, 1239–1257. [\[CrossRef\]](#)
43. Liu, G.; Zou, J. Level set evolution with sparsity constraint for object extraction. *IET Image Process.* **2018**, *12*, 1413–1422. [\[CrossRef\]](#)
44. Wu, G.; Pedrycz, W.; Suganthan, P.N.; Li, H. Using variable reduction strategy to accelerate evolutionary optimization. *Appl. Soft Comput.* **2017**, *61*, 283–293. [\[CrossRef\]](#)
45. Li, R.; Zheng, Q.; Li, X.; Yan, Z. Multi-objective optimization for rebalancing virtual machine placement. *Future Gener. Comput. Syst.* **2020**, *105*, 824–842. [\[CrossRef\]](#)
46. Srinivas, N.; Deb, K. Multiobjective optimization using nondominated sorting in genetic algorithms. *Evol. Comput.* **1994**, *2*, 221–248. [\[CrossRef\]](#)
47. Coello, C.A.C.; Pulido, G.T.; Lechuga, M.S. Handling multiple objectives with particle swarm optimization. *IEEE Trans. Evol. Comput.* **2004**, *8*, 256–279. [\[CrossRef\]](#)
48. Felde, I.; Szénási, S. Estimation of temporospatial boundary conditions using a particle swarm optimisation technique. *Int. J. Microstruct. Mater. Prop.* **2016**, *11*, 288–300. [\[CrossRef\]](#)
49. Wang, F.; Zhang, H.; Zhou, A. A particle swarm optimization algorithm for mixed-variable optimization problems. *Swarm Evol. Comput.* **2021**, *60*, 100808. [\[CrossRef\]](#)
50. Deb, K.; Pratap, A.; Agarwal, S.; Meyarivan, T. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evol. Comput.* **2002**, *6*, 182–197. [\[CrossRef\]](#)
51. Yi, J.H.; Deb, S.; Dong, J.; Alavi, A.H.; Wang, G.G. An improved NSGA-III algorithm with adaptive mutation operator for Big Data optimization problems. *Future Gener. Comput. Syst.* **2018**, *88*, 571–585. [\[CrossRef\]](#)
52. Yi, J.H.; Xing, L.N.; Wang, G.G.; Dong, J.; Vasilakos, A.V.; Alavi, A.H.; Wang, L. Behavior of crossover operators in NSGA-III for large-scale optimization problems. *Inf. Sci.* **2020**, *509*, 470–487. [\[CrossRef\]](#)
53. Zhang, Q.; Li, H. MOEA/D: A multiobjective evolutionary algorithm based on decomposition. *IEEE Trans. Evol. Comput.* **2007**, *11*, 712–731. [\[CrossRef\]](#)
54. Wang, R.; Zhang, Q.; Zhang, T. Decomposition-based algorithms using pareto adaptive scalarizing methods. *IEEE Trans. Evol. Comput.* **2016**, *20*, 821–837. [\[CrossRef\]](#)
55. Wang, R.; Zhou, Z.; Ishibuchi, H.; Liao, T.; Zhang, T. Localized weighted sum method for many-objective optimization. *IEEE Trans. Evol. Comput.* **2018**, *22*, 3–18. [\[CrossRef\]](#)
56. Wang, G.G.; Tan, Y. Improving metaheuristic algorithms with information feedback models. *IEEE Trans. Cybern.* **2017**, *49*, 542–555. [\[CrossRef\]](#)
57. Ishibuchi, H.; Tsukamoto, N.; Nojima, Y. Evolutionary many-objective optimization: A short review. In Proceedings of the 2008 IEEE Congress on Evolutionary Computation (CEC 2008), Hong Kong, China, 1–6 June 2008; pp. 2419–2426. [\[CrossRef\]](#)
58. Petsios, T.; Zhao, J.; Keromytis, A.D.; Jana, S. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017), Dallas, TX, USA, 30 October–3 November 2017; pp. 2155–2168. [\[CrossRef\]](#)
59. Wang, H.; Xie, X.; Li, Y.; Wen, C.; Li, Y.; Liu, Y.; Qin, S.; Chen, H.; Sui, Y. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In Proceedings of the 42nd International Conference on Software Engineering (ICSE 2020), Han River, Seoul, Korea, 6–11 July 2020; pp. 999–1010. [\[CrossRef\]](#)
60. Gan, S.; Zhang, C.; Chen, P.; Zhao, B.; Qin, X.; Wu, D.; Chen, Z. GREYONE: Data flow sensitive fuzzing. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 2020), Boston, MA, USA, 12–14 August 2020; pp. 2577–2594.
61. Wang, J.; Chen, B.; Wei, L.; Liu, Y. Superior: Grammar-aware greybox fuzzing. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE 2019), Montreal, QC, Canada, 25–31 May 2019; pp. 724–735. [\[CrossRef\]](#)
62. Wang, Y.; Jia, X.; Liu, Y.; Zeng, K.; Bao, T.; Wu, D.; Su, P. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS 2020), San Diego, CA, USA, 23–26 February 2020; pp. 1–17. [\[CrossRef\]](#)
63. Lattner, C.; Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In Proceedings of the International Symposium on Code Generation and Optimization (CGO 2004), Palo Alto, CA, USA, 20–24 March 2004; pp. 75–86. [\[CrossRef\]](#)
64. Jasper. Available online: <https://www.ece.uvic.ca/~frodo/jasper/> (accessed on 23 September 2020).
65. Libsass. Available online: <https://sass-lang.com/libsass> (accessed on 23 September 2020).
66. Exiv2. Available online: <https://exiv2.org/> (accessed on 23 September 2020).
67. Ming. Available online: <https://github.com/libming/libming> (accessed on 23 September 2020).

- 
68. Openjpeg. Available online: <https://www.openjpeg.org/> (accessed on 23 September 2020).
  69. Bento4. Available online: <https://www.bento4.com/> (accessed on 23 September 2020).
  70. Binutils. Available online: <https://www.gnu.org/software/binutils/> (accessed on 23 September 2020).
  71. Afl-cov. Available online: <https://github.com/soh0ro0t/afl-cov> (accessed on 23 September 2020).
  72. Serebryany, K.; Bruening, D.; Potapenko, A.; Vyukov, D. AddressSanitizer: A fast address sanity checker. In Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC 2012), Boston, MA, USA, 13–15 June 2012; pp. 309–318.