Alphuzz: Monte Carlo Search on Seed-Mutation Tree for Coverage-Guided Fuzzing

Yiru Zhao
zhaoyiru@whu.edu.cn
Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University
Wuhan, China

Xiaoke Wang
xkernel@whu.edu.cn
Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University
Wuhan, China

Lei Zhao*
leizhao@whu.edu.cn
Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University
Wuhan, China

Yueqiang Cheng
yueqiang.cheng@nio.io
Nio security
Mountain View, CA, USA

Heng Yin
heng@cs.ucr.edu
University of California, Riverside
Riverside, CA, USA

ABSTRACT

Coverage-based greybox fuzzing (CGF) has been approved to be effective in finding security vulnerabilities. Seed scheduling, the process of selecting an input as the seed from the seed pool for the next fuzzing iteration, plays a central role in CGF. Although numerous seed scheduling strategies have been proposed, most of them treat these seeds independently and do not explicitly consider the relationships among seeds.

In this study, we make a key observation that the relationships among seeds are valuable for seed scheduling. We design and propose a "seed mutation tree" by investigating and leveraging the mutation relationships among seeds. With the "seed mutation tree", we further model the seed scheduling problem as a Monte-Carlo Tree Search (MCTS) problem. That is, we select the next seed for fuzzing by walking this "seed mutation tree" through an optimal path, based on the estimation of MCTS. We implement two prototypes, Alphuzz on top of AFL and Alphuzz++ on top of AFL++. The evaluation results on three datasets (the UniFuzz dataset, the CGC binaries, and 12 real-world binaries) show that Alphuzz and Alphuzz++ outperform state-of-the-art fuzzers with higher code coverage and more discovered vulnerabilities. In particular, Alphuzz discovers 3 new vulnerabilities with CVEs.

*Lei Zhao is the corresponding author. This work is partly supported by National Natural Science Foundation of China under Grant No.62172305.

ACSAC '22, December 5-9, 2022, Austin, Texas, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-8579-4/21/12…$15.00
https://doi.org/10.1145/3564625.3564660

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**.

KEYWORDS

Fuzzing, Seed scheduling strategy, Vulnerability detection

ACM Reference Format:
Yiru Zhao, Xiaoke Wang, Lei Zhao, Yueqiang Cheng, and Heng Yin. 2022. Alphuzz: Monte Carlo Search on Seed-Mutation Tree for Coverage-Guided Fuzzing. In *Annual Computer Security Applications Conference (ACSAC '22).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3564625.3564660

1 INTRODUCTION

Coverage-based greybox fuzzing (CGF) has been approved to be very effective in finding security vulnerabilities in real-world applications and has been widely studied and used in both academia and industry [6, 17, 24, 36, 46, 47]. In general, CGF starts with several initial inputs (a.k.a. seed inputs). Then, CGF leverages a seed scheduling strategy to select an input as the seed and generates new inputs by randomly mutating the seed. With lightweight program instrumentation, CGF executes a program with these newly generated inputs and collects code coverage to further guide the next cycle of seed scheduling. In this way, CGF makes progress in discovering vulnerabilities through iterative seed scheduling, mutation, and program state exploration.

Seed scheduling, the process of selecting a seed from the seed pool for the next fuzzing iteration, plays a central role in CGF. First, CGF utilizes a seed scheduling strategy to bridge the gap between limited fuzzing effort and an ever-increasing number of seeds. More importantly, seed scheduling determines the directions of exploring program states. Specifically, CGF explores program states by dynamic execution using the inputs mutated from seeds. As mutations are often slight (e.g., bit-flip or byte-flip in AFL [47]), the newly generated input often differs from the seed only in small input regions. Consequently, the execution using the new input could result in a transition to a path that is a neighbor of the path corresponding to the seed. In other words, dynamic executions using mutated inputs can be regarded as the process of exploring the neighbor paths of the corresponding path of the seed. Therefore, seed scheduling refers to the selection of an execution path for exploring its neighbor paths, which determines the directions of exploring program states.

Numerous seed scheduling strategies have been proposed. Generally speaking, these strategies aim to assign a score to each seed based on certain criteria and then choose the seed with the highest score for the next fuzzing iteration. For example, AFL [47] prefers seeds with the smallest size and shortest execution time. AFLFast [6] assigns more energy to the seed with low frequency. FairFuzz [24] prefers to choose a seed that covers more rare branches. EcoFuzz [46] favors newly generated seeds. These strategies treat these seeds independently and do not explicitly consider the relationships among the seeds.

In this paper, we make a key observation that the relationships among seeds are valuable for seed scheduling, especially the seed mutation relationship (e.g., "$A \longrightarrow B$" means Seed B is mutated from Seed A). Furthermore, a tree structure can be formed for these seeds by following their mutation relationships, which we refer to as the "seed mutation tree". Then we model the seed scheduling problem as a Monte-Carlo Tree Search (MCTS) problem. That is, we select the next seed for fuzzing by walking this seed mutation tree through an optimal path, based on the estimation of MCTS.

The main advantage of the "seed mutation tree" is that it benefits the seed scheduling to balance between exploitation (i.e., repeatedly exercising a high-potential seed and its neighbors) and exploration (i.e., trying a rarely-exercised seed) for optimal performance. As demonstrated above, seed scheduling refers to the selection of an execution path for exploring its neighbor paths. With the relationships among seeds, we can further organize paths corresponding to every seed as a tree structure, which is denoted as execution tree [10]. In this way, we can model the fuzzing process of exploring program states as the growth of the execution tree. Based on this observation, we can leverage the "seed mutation tree" to approximate the execution tree, in which each path corresponds to a seed. For example, the mutation relationship ("$A \longrightarrow B$") indicates that the CGF covers a new path corresponding to B by converting the result of a conditional jump on the path of A. As an approximation to the execution tree, the "seed mutation tree" enables the seed scheduling to balance exploitation (the tree's depth) and exploration (the tree's width), which indicates the fuzzing progress within a specific direction and multiple directions, respectively.

Moreover, the construction of "seed mutation tree" is lightweight, because mutation relationships are readily available in CGF and do not require extra instrumentation and expensive computation.

A recent work AFL-HIER [44] also leverages a tree structure to manage all the seeds and uses the UCB algorithm to perform seed scheduling. Our approach differs from AFL-HIER in three aspects. First, the insights are different. AFL-HIER defines different coverage sensitivity metrics and clusters seeds to reduce the impact of similarity. The core of our approach is to construct a seed tree using the mutation relationships, which is a lightweight and practical approximation to the execution tree. Second, the tree structures are different. In AFL-HIER, every internal node means a cluster of seeds that have the same coarse-grained coverage measurement. Every leaf node represents one seed. In our approach, every node in the tree refers to one seed, and the edge represents the mutation relationship between seeds. Third, the performances of seed scheduling are different. As AFL-HIER requires maintaining multiple coverage metrics for clustering, it introduces overhead to seed scheduling and finally results in a negative impact on the fuzzing throughput. By contrast, seed scheduling on our seed tree is lightweight. As throughput is a significant factor for the fuzzing performance, our approach can outperform AFL-HIER.

We implement two prototypes of the MCTS-based seed scheduling, Alphuzz on top of AFL [47], and Alphuzz++ on top of AFL++[15]. To demonstrate the effectiveness of our approach, we conduct a comprehensive evaluation on three datasets, including UniFuzz [25], CGC binaries [13], and 12 real-world binaries. We compare the performance of Alphuzz with AFL-based techniques AFL, AFLFast, FairFuzz, and EcoFuzz. For AFL++-based techniques, we compare Alphuzz++ with AFL++ and AFL++-HIER. Evaluation results show that Alphuzz and Alphuzz++ outperform other baseline techniques in terms of code coverage and discover more vulnerabilities. Specifically, Alphuzz achieves higher code coverage on UniFuzz than AFL, AFLFast, FairFuzz, and EcoFuzz on 14, 15, 14 and 16 out of 18 binaries, respectively. Alphuzz and Alphuzz++ discover more unique bugs and exploitable vulnerabilities than others. For CGC binaries, Alphuzz discovers 87 vulnerabilities, whereas AFL, AFLFast, EcoFuzz, and FairFuzz discover 83, 83, 73, and 76 respectively. In addition, Alphuzz discovers 3 new CVEs on 12-real-world binaries dataset.

The contributions of this study are as follows:

- ***New insight***. We make a key observation that the relationships among seeds are valuable for seed scheduling. We investigate and leverage the seed mutation relationships to construct a "seed mutation tree", which is an approximation of the execution tree for the fuzzing progress and can further benefit the seed scheduling to balance exploitation and exploration.
- ***New fuzzing technology***. With the "seed mutation tree", we model the seed scheduling problem as a Monte-Carlo Tree Search (MCTS) problem and propose an MCTS-based seed scheduling strategy. This strategy strikes a balance between exploitation and exploration, due to the nature of the MCTS algorithm.
- ***Open-source implementation***. We implement two prototypes, Alphuzz and Alphuzz++, and conduct comprehensive evaluations to demonstrate their performance. Results show that Alphuzz and Alphuzz++ outperform state-of-the-art fuzzers with more code coverage and more vulnerabilities discovered. The source code is available at https://github.com/zzyyrr/Alphuzz_overview.git.

2 MOTIVATION AND INSIGHT

In this section, we first illustrate our motivation by discussing the limitations of existing seed scheduling strategies with an example and then state our insight.

2.1 Motivating example

We use a piece of code in Figure 1 as an example to illustrate our motivation. As shown in Figure 1, b_i represents the basic block in this code, and $\langle b_i, b_j \rangle$ represents the branch from b_i to b_j. In column 3 to 7, t_1 to t_5 are test cases retained by CGF. The string

	code	test cases				
		t_1	t_2	t_3	t_4	t_5
		'x'	'y'	'xaf'	'xc'	'ym'
	int func1 (input) {					
b_0	if (input[0] == 'x')	●	●	●	●	●
b_1	func2(input);	●		●	●	
b_2	else if (input[0] == 'y')		●			●
b_3	func3(input);		●			●
	}					
	int func2 (input) {					
b_4	if (input[1] == 'a')	●		●	●	
b_5	if (input[2] == 'f')			●		
b_6	…			●		
b_7	else if (input[1] == 'b')	●				
b_8	…					
b_9	else if (input[1] == 'c')	●			●	
b_{10}	…				●	
	}					
	int func3 (input) {					
b_{11}	if (input[1] == 'm')		●			●
b_{12}	if (input[2] == 'n')					●
b_{13}	func4 (input);					
b_{14}	else exit (0);		●			
	}					

Figure 1: Motivating example.

below each test case is the content of the input file. For example, the content of test case t_1 is $'x'$. When executing with different test cases, different basic blocks are covered. The solid circle indicates that the test case covers the corresponding basic block.

In order to describe the seed scheduling process clearly, we suppose that the CGF starts with two initial test cases, t_1 and t_2. After several fuzzing iterations, CGF generates and retains three new test cases. Test case t_3 and t_4 are generated via performing mutations on test case t_1. Similarly, test case t_5 is derived from t_2.

With these test cases in the seed pool, lots of seed scheduling strategies are proposed to guide the direction of CGF. Figure 2 shows how existing seed scheduling strategies manage the test cases in Figure 1. To better understand their principles, Figure 2 (a) shows the execution tree according to Figure 1. Then we distinguish the paths covered by test cases with different colors. For example, the path in blue color is covered by test case t_3.

As shown in Figure 2 (b), AFL [47], AFLFast [6], FairFuzz [24], and EcoFuzz [46] propose different criteria for seed scheduling, but they all treat the seeds independently and only consider the individual features of seeds. AFL reserves the seed with the smallest size and shortest execution time for every covered branch to test as many inputs as possible in a limited time. Since the seed t_3 covers the new branch $\langle b_4, b_5\rangle$ and $\langle b_5, b_6\rangle$, and t_4 covers the new branch $\langle b_4, b_9\rangle$ and $\langle b_9, b_{10}\rangle$, they are both considered interesting, even though they both cover branch $\langle b_0, b_1\rangle$ and $\langle b_1, b_4\rangle$. AFLFast prefers low-frequency paths which fuzzing spent less time and effort exploring them than high-frequency paths. Thus, AFLFast chooses the paths covered by newly generated seeds t_3, t_4, and t_5. FairFuzz calculates the times each branch has been covered and selects the seed that covers more low-frequency branches. So, seeds t_1, t_3, and t_4 are selected by FairFuzz. EcoFuzz gives the highest priority to unfuzzed seeds. So EcoFuzz selects seeds t_3, t_4, and t_5.

The seed scheduling strategies above are unaware of the relationships among seeds. Seeds covering new branches are always given high priority by the above seed selection strategies, regardless of weather the seeds are concentrated in the same code region. For example, seeds t_3 and t_4 lead fuzzing to constantly explore the region of the *func2*. As a result, as long as new branches are consistently covered in this area of code, fuzzing will focus on this code region, and be difficult to jump out and explore other areas that may have potential vulnerabilities.

AFL-HIER [44] defines many coverage metrics of different sensitivities to cluster the similar seeds together. As shown in Figure 2 (c), AFL-HIER uses function coverage and edge coverage to cluster the seeds together. For example, seeds t_1, t_3, and t_4 are under the same node F_1,F_2 because they cover the same functions. When performing seed scheduling, AFL-HIER starts from the nodes at function coverage level, and calculates scores for nodes based on their function-level rareness and their rewards. The rewards of the cluster are the average rewards of all the seeds in this cluster. However, elder seeds and newly generated seeds are clustered under the same nodes. The rewards of newly generated seeds can be weakened by the fuzzing effort spent on elder seeds. Therefore, the individual feature of seeds is flattened, which brings a negative impact on fuzzing deeper paths.

As discussed above, the seed scheduling strategy should consider both the relationships among seeds and the individual features of seeds. Therefore, we propose a new seed scheduling strategies which satisfy these two requirements.

2.2 Our insight

In this paper, we consider the seed scheduling as selecting an optimal path in order to explore its neighbor paths. As discussed above, different paths in the execution tree lead fuzzing to explore different directions. With the execution tree growing, some paths tend to focus on the same region of the program, while some paths tend to explore different regions. The paths covered by seeds with mutation relationships share the same direction on the execution tree in the beginning, and then separate in different directions. Therefore, we can leverage the mutation relationships among seeds to construct a "seed mutation tree", which can be an approximation of the execution tree. As shown in Figure 2 (d), the relationships among different fuzzing directions are reserved in our "seed mutation tree". For example, t_1, t_3, and t_4 are in the same sub-tree, because they all covered basic block b_0, b_1 and b_4.

In the meantime, the growth of the execution tree can also be represented in "seed mutation tree", where newly generated seeds are always leaf nodes and elder seeds are usually internal nodes. Thus, the hierarchy of nodes in the tree can show the effort fuzzing spent on different paths, which is an important individual feature in seed scheduling, but is flattened in AFL-HIER.

With "seed mutation tree", the seed scheduling can be regarded as a process of searching for an optimal seed in the tree. We further model the seed scheduling as a Monte Carlo Tree Search (MCTS) problem and propose an MCTS-based seed scheduling strategy. The following sections give the details of our "seed mutation tree" and MCTS-based seed scheduling strategy.

3 SEED MUTATION TREE

In this section, we first present the definition and the construction of the "seed mutation tree". Then, we discuss the challenges of performing seed scheduling on it.

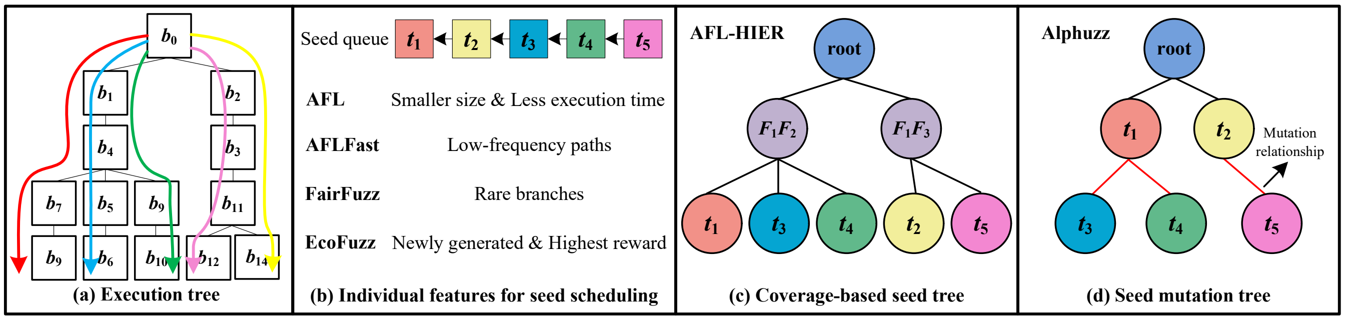
(a) Execution tree
Seed queue
AFL Smaller size & Less execution time
AFLFast Low-frequency paths
FairFuzz Rare branches
EcoFuzz Newly generated & Highest reward
(b) Individual features for seed scheduling
AFL-HIER
(c) Coverage-based seed tree
Alphuzz
Mutation relationship
(d) Seed mutation tree

Figure 2: Existing seed scheduling strategies.

3.1 Definition

We leverage the mutation relationships among seeds to construct the "seed mutation tree", which is defined as follows.

Definition 1. A "seed mutation tree" is a directed tree $T = (V, E, \alpha)$, where:

- Each element v in the set of vertices V corresponds to a seed;
- Each element e in the set of edges $E \subseteq V \times V$ corresponds to the *mutation relationship* between two vertices v and w.
- The labeling function $\alpha : E \longrightarrow \Sigma$ associates edges among seeds in terms of their mutation relationships.

3.2 Tree construction

According to **Definition 1**, we construct a "seed mutation tree" during the fuzzing process.

First, before the fuzzing iteration begins, we construct the initial tree structure with the initial seeds and an auxiliary root node. The auxiliary root node is the parent node of all the initial seeds for there might be more than one initial seed. As shown in Figure 2 (d), the root node, node t_1 and node t_2 form the initial tree structure.

Then, CGF selects a seed as the base of mutation according to the seed scheduling strategy, and performs mutations on it to generate new test cases. Next, newly generated test cases that cover new paths are added to the "seed mutation tree" as the child of the original seed. For example, CGF performs random mutations on seed t_1, and generates test case t_3 and t_4. They cover two new paths, t_3 covers the blue path and t_4 covers the green path, respectively. We add the node t_3 to the tree as the child node of t_1 based on the mutation relationship between them, and node t_4 the same.

With CGF continuing to explore the targeted application, we continue to add the seeds covering new paths as nodes to the tree structure according to the mutation relationship between the seeds.

Some special mutations may operate on two seeds. For example, the mutation of splice in AFL will splice a seed with a second seed to generate a new input. In such cases, we only construct an edge between the new node and the node corresponding to the first seed, because the first seed is selected by the seed scheduling strategy, whereas the second one is randomly selected for mutations.

3.3 Challenges of seed scheduling on the tree

The seed scheduling can be regarded as a process of searching for an optimal seed in the "seed mutation tree". However, performing seed scheduling on the tree faces the following three challenges.

Challenge 1. The search space is large and ever-increasing. As new seeds are constantly being added to the tree, it is time-consuming and tedious to traverse every node in the tree to select the best seed. Therefore, the "seed mutation tree" cannot be fully searched. Thus, we need a heuristic algorithm to provide the current best decision.
Challenge 2. The core of the heuristic algorithm is to assign scores to seeds. However, the score of the path covered by a seed may decrease as fuzzing explores its adjacent paths more and more thoroughly. Thus, it is essential yet challenging to update the scores of all the seeds on the tree after every fuzzing iteration.
Challenge 3. It is difficult to balance exploration and exploitation. With the impact of randomization character of fuzzing in nature, the scores of seeds calculated after one fuzzing iteration are affected by such uncertainties. Thus, how to balance exploration and exploitation under these uncertain factors is challenging.

4 MONTE CARLO SEARCH ON THE SEED-MUTATION TREE

To tackle the above challenges, we propose an MCTS-based seed scheduling strategy. In this section, we first introduce the MCTS algorithm. Then, we present the design of our MCTS-based seed scheduling strategy.

4.1 Monte carlo tree search

MCTS [4, 7, 12, 22, 26] is an algorithm for taking optimal decisions through sequentially built trees based on random sampling. The process consists of four steps:

Selection. Using a specific tree policy, MCTS starts from the root node R, recursively selects optimal child nodes until a leaf node L is reached.

Expansion. If L is not a terminal node, then MCTS creates one or more child nodes. Further, it selects one node C from these child nodes. In this step, child nodes refer to any valid moves from the state defined by L.

Simulation. The simulation is performed by randomly choosing moves until a result or a predefined state is achieved.

Back propagation. This step back propagates from the new node C to the root node R, and updates the simulation results.

Upper confidence bounds for trees. The main challenge in a planning problem is to balance exploitation and exploration. For

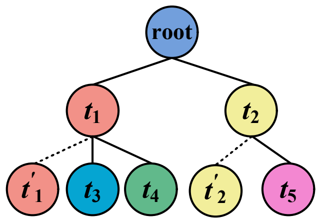

Figure 3: "Seed mutation tree" with variants.

this challenge, the Upper Confidence Bounds (UCB) algorithm [3] is typically adopted in MCTS.

$$UCB = v_i + k\sqrt{\frac{lnN}{n_i}} \quad (1)$$

The formula of UCB is shown in **Formula 1**, where v_i is the estimated value of the node, n_i is the number of the times the node has been visited, N is the total number of times that its parent has been visited, and k is a tunable bias parameter.

The UCB formula balances the exploitation of known rewards with the exploration of relatively unvisited nodes for optimal performance. When applied in MCTS, the UCB formula is extended to the tree search and named as the Upper Confidence Bounds for Trees (UCT) [22]. That is, UCT is a special case for UCB in MCTS.

4.2 MCTS-based seed scheduling

Inspired by the MCTS algorithm, we propose an MCTS-based seed scheduling strategy as illustrated in Algorithm 1.

Before fuzzing iteration begins, we construct the initial tree structure with initial seeds. Then, MCTS starts from the root node and, at each level, selects the next node with the highest score until a leaf node is reached. To calculate the scores of seeds, we leverage the UCT algorithm with fuzzing rewards and fuzzing effort to balance exploration and exploitation (section 4.4). Due to UCT algorithm and random sampling, we can search the tree in an asymmetric fashion in contrast to traditional tree search methods such as minimax. Thus, we can reduce the impact of uncertainties.

Afterwards, fuzzing randomly mutates the selected seed while monitoring the executions of newly generated inputs. we regard these processes as the simulation of the selected seed. Since not all newly generated inputs improve code coverage, we perform simulation first, and only add inputs that increase code coverage to the tree. Finally, we update the scores of the seeds after each fuzzing iteration (section 4.5).

4.3 Tree construction and expansion

As fuzzing progresses, seeds are added to the tree according to the mutation relationships. In our "seed mutation tree", there are two different roles for an internal node. First, it refers to a seed. Second, it is also the root node of a sub-tree, from the perspective of the tree structure. Therefore, an internal node can also be selected as a seed even if it has multiple child nodes. However, the MCTS algorithm always selects the leaf node.

More important, these two different roles indicate different fuzzing scores. As a seed, it refers to the fuzzing score for selecting this seed. As the root node of a sub-tree, it refers to the summary fuzzing score for selecting every seed in the sub-tree. With the impact of such two different roles, the calculations of fuzzing score for an internal node will be also different.

Algorithm 1: MCTS-based Seed Scheduling.

```
/* Initialization */
1  $T \leftarrow$ Init_tree(initial seeds)
2  while not EndConditions() do
       /* Selection */
3      $v \leftarrow root\_node$
4      while $v$ is not a leaf node do
5          $v \leftarrow \underset{v' \in v.child}{argmax} \frac{Q(v')}{N(v')} + k\sqrt{\frac{lnN(v)}{N(v')}}$
6      end
       /* Simulation */
7      $\{S\} \leftarrow Fuzz\_one(v.input)$
       /* Expansion */
8      for $s$ in $\{S\}$ do
9          $v$.add_child_node($s$)
10     end
       /* Back propagation */
11     while $v$ is not null do
12         $N(v) \leftarrow N(v) + 1$
13         $Q(v) \leftarrow Q(v) + \Delta Q(\{S\})$
14         $v \leftarrow v.parent\_node$
15     end
16 end
```

To address this incompatibility, we update the "seed mutation tree" by inserting a variant node for every internal node. The internal node refers to the root node of its sub-tree, and the variant refers to the corresponding seed. In this way, the variant will be a leaf node of the internal node. Regarding the fuzzing scores, the internal node refers to the summary fuzzing scores for every seed in the sub-tree. By contrast, the variant refers to the fuzzing score for the corresponding seed.

Let us take the motivating example for illustration. As show in Figure 3, t_1 has two child nodes, t_3 and t_4. t_2 has one child node t_5. To distinguish between an internal node itself and its sub-tree, we construct two variants for both t_1 and t_2, which are denoted as t'_1 and t'_2, respectively.

With variants, the construction of the "seed mutation tree" requires updating for internal nodes. Whenever a leaf node is generated from a node without a variant node, a variant of the parent node will also be created. Then, both the leaf node and the variant node will be inserted into the tree as the child nodes of the parent node.

The design of variants addresses the problem of incompatibility between MCTS and "seed mutation tree". By setting a variant for every internal node, the variant node will be a leaf node for the internal node. With this design, MCTS is able to search for a promising node until a leaf node is reached.

4.4 Calculation of seed score

We leverage the UCT algorithm to calculate the score of a seed in terms of fuzzing effort and rewards. We evaluate the fuzzing effort

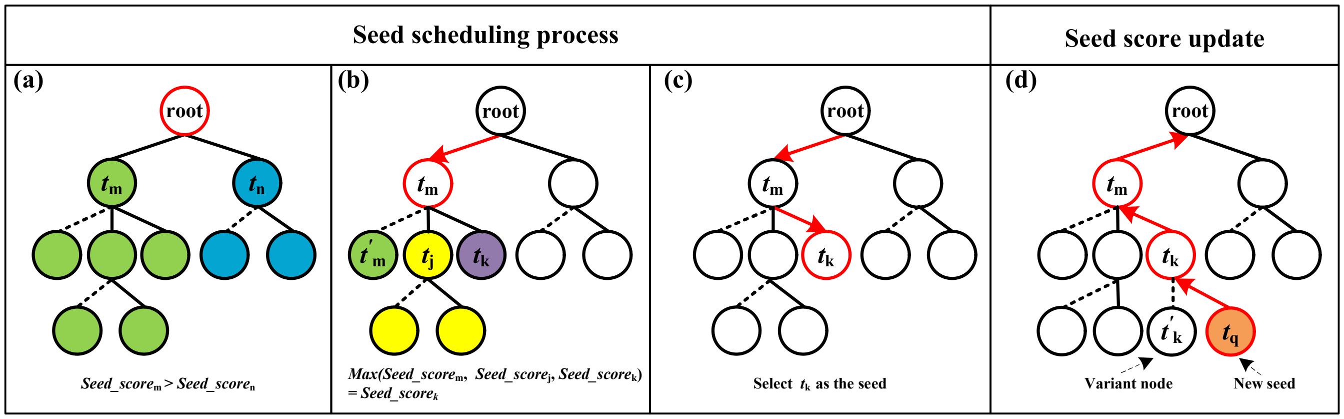

Figure 4: MCTS-based seed scheduling process.

that cost by fuzzing the same seed rapidly and calculate the rewards of a scheduled seed in terms of new code coverage traversed by the mutations based on it. With these two factors, fuzzing effort and fuzzing rewards, the calculation of *seed_score* is designed as shown in **Formula 2**.

$$seed\_score = \frac{q_i}{n_i} + k\sqrt{\frac{lnN_i}{n_i}} \quad (2)$$

In **Formula 2**, $\frac{q_i}{n_i}$ represents the rewards of this seed. The $k\sqrt{\frac{lnN_i}{n_i}}$ represents the fuzzing effort spent on this seed.

In $\frac{q_i}{n_i}$, n_i refers to the number of times this node has been selected. q_i represents the number of *unique branches* covered by the seed. Specifically, most of the existing fuzzers leverage branch coverage to guide the evolution. Therefore, for each seed, we collect the branches covered by the seed as its *branch set*. For the internal node, we collect the branches covered by this node and all its descendants as its *branch set*. We then put these *branch sets* together and calculate the number of *unique branches* for each child node. The number of *unique branches* means the number of branches that are only covered by the child node compared to other sibling nodes. Then, we assign the number of *unique branches* to the value of q_i.

In $k\sqrt{\frac{lnN_i}{n_i}}$, n_i refers to the number of times this node has been selected. N_i represents the number of times the parent of the node has been scheduled. k is a constant, which controls the balance between exploration and exploitation. To investigate the impact of different values of k on seed scheduling, we conduct an experiment which is shown in **section 5.8**.

The selection process of our seed scheduling strategy is shown in Figure 4. As shown in Figure 4 (a), each fuzzing iteration starts from the root node. Then, we calculate scores for every child node of the root node. By comparing scores among sibling nodes, our MCTS-based seed scheduling strategy will recursively select the promising one with the highest score, until a leaf node is reached. As shown in Figure 4 (b), we assign scores to t'_m, t_j and t_k. The score of t_k is the highest among these three child nodes. Therefore, t_k becomes the currently selected node. Then, as shown in Figure 4 (c), t_k is a leaf node, therefore the seed scheduling stops and selects t_k as the base for the mutation stage of fuzzing.

4.5 Seed score update

After each round of seed scheduling, we calculate the rewards and effort of this fuzzing iteration and update the scores of seeds.

Specifically, to update the rewards, as shown in the Figure 4 (d) , when a new node, t_q for example, is added to a sub-tree, we update the information of newly covered branches of t_q from t_q to all its parent nodes t_k and t_m.

To update the fuzzing effort, the selected times of the selected node and all its parent nodes are increased by one. That is to say, the selected times of t_k and t_m are increased by 1. We update the rewards and effort from the selected seed back to the root node. This process goes in exactly the opposite direction to seed scheduling. In this way, we are able to record the rewards and effort of this time's seed scheduling to guide the next iteration.

5 EVALUATION

We implement two prototypes, Alphuzz and Alphuzz++, on top of AFL [47] and AFL++ [15], respectively. To demonstrate the effectiveness of our approach, we conduct a comprehensive evaluation by comparing our approach with the state-of-the-art fuzzing techniques, with respect to code coverage and vulnerability detection.

5.1 Datasets

We leverage three datasets: the Cyber Grand Challenge (CGC) dataset [13], UniFuzz [25], and 12 real-world binaries.

The CGC dataset [13] includes binaries from the CGC Qualifying Event and the CGC Final Event, which are widely used in previous techniques [2, 44]. Every CGC binary is injected with one or more memory corruption vulnerabilities. In our evaluation, we exclude programs involving communication with multiple programs, and programs on which AFL cannot work. In total, we use 188 CGC binaries for evaluation. As the original CGC binaries are customized with specific syscalls, we leverage their compatible Linux versions [31] for evaluation.

UniFuzz [25] is an open-source and pragmatic metrics-driven platform for evaluating fuzzing techniques, which consists of 20 programs. Unfortunately, *sqlite3* and *ffmpeg* failed to run correctly due to frequent timeouts during our evaluation. Thus, we only leverage 18 programs.

Table 1: Real-world binaries evaluated in our experiments.

Program	Input	cmd line	Program	Input	cmd line
cjpeg	bmp	@@	infotocap	text	@@
exiv2	jpg	@@ /dev/null	mp3gain	mp3	@@
nm	elf	-AD @@	objdump	elf	-d @@
pdfimages	pdf	@@ /dev/null	pngfix	png	@@
readelf	elf	-a @@	size	elf	-At @@
tiff2pdf	tiff	@@	xmlwf	xml	@@

We then choose 12 real-world binaries based on the following features: popularity, frequency of being tested, and diversity of categories. As shown in Table 1, these 12 real-world binaries include popular tools (e.g., *nm*, *objdump*), image processing libraries (e.g., *libjpeg*, *libtiff*), terminal processing libraries (e.g., *ncurses*), and document processing libraries (e.g., *xpdf*), etc.

5.2 Baseline techniques

In recent years, many fuzzing techniques have been proposed to improve the performance of fuzzing from different aspects, including coverage sensitivity, mutation algorithms, input generation, and execution monitoring [8, 18, 28, 30, 42, 45, 48]. As Alphuzz proposes a new seed scheduling strategy, we only select fuzzing techniques focusing on seed scheduling algorithms as baseline techniques. Furthermore, we exclude fuzzing techniques requiring the source code of the targeted binaries. For AFL-based fuzzing techniques, we choose AFL [47], AFLFast [6], FairFuzz [24] and EcoFuzz [46]. For AFL++-based fuzzing techniques, we choose AFL++ and AFL++-HIRE as baseline techniques.

5.3 Experiment setup

We run the experiments on a server configured with 40 CPU cores of 2.50GHz E5-2670 v2, 125GB RAM, and running on the 64-bit Ubuntu 16.04 LTS.

We fuzz each binary for 24 hours as done in previous studies [6, 24]. To alleviate the impact of the randomness in fuzzing, we run each experiment for 10 rounds, and report the statistical results for a more comprehensive evaluation.

All the experiments are based on the QEMU-mode in AFL (so as to support binary-only targets) and configured with the same initial seeds and instructions. The initial seeds for CGC binaries come from the examples provided by the CGC challenges [13]. The initial seeds for UniFuzz are provided by UniFuzz [25]. The initial seeds for the 12 real-world binaries come from the default seed examples provided by AFL.

5.4 Vulnerability detection

To measure the vulnerability detection capability of Alphuzz, we analyse the experimental results on CGC and UniFuzz. As CGC binaries are manually designed and embedded with known bugs, we mainly evaluate the number of unique bugs. For UniFuzz, we evaluate the number of unique bugs and the number of unique exploitable vulnerabilities.

5.4.1 *Unique bugs on CGC dataset*.

To summarize the evaluation results of the 10-round experiments, we count the number of times each bug is found in 10-round experiments. Then we count the number of bugs in terms of the number of times the bug is found in 10-round experiments by each baseline technique. As shown in Table 2, columns 2-11 list the number of bugs that are discovered in 10 rounds by different metrics. For example, column 2 reports the number of vulnerabilities that are discovered in every round. Column 3 reports the number of vulnerabilities that are discovered in at least 9 rounds, and so on.

Table 2: Numbers of discovered bugs on CGC.

Fuzzer	= 10	≥ 9	≥ 8	≥ 7	≥ 6	≥ 5	≥ 4	≥ 3	≥ 2	≥ 1
Alphuzz	70	72	72	73	75	75	80	83	87	87
AFL	69	69	69	70	71	75	78	83	83	83
AFLFast	67	67	70	71	73	77	79	79	81	83
EcoFuzz	66	66	66	67	68	68	68	68	73	73
FairFuzz	67	67	68	70	71	72	72	73	74	76
Alphuzz++	72	73	75	75	77	80	80	85	88	88
AFL++	70	72	75	75	75	76	79	83	83	84
AFL++-Hier	72	73	73	75	76	80	83	83	85	85

Table 3: Numbers of discovered unique bugs.

Binary	AFL	AFLFast	EcoFuzz	FairFuzz	Alphuzz
cflow	3	4	2	3	4
exiv2	0	1	1	1	1
flvmeta	3	3	3	4	3
gdk	6	6	2	0	8
imginfo	0	1	0	0	1
infotocap	2	3	0	2	2
jhead	0	0	2	0	0
jq	0	0	0	0	0
lame	3	2	2	3	3
mp3gain	6	7	4	3	7
mp42aac	2	2	2	2	2
mujs	0	0	0	0	1
nm	0	0	0	0	0
objdump	3	3	2	1	4
pdftotext	1	1	0	1	8
tcpdump	0	0	0	0	0
tiffsplit	4	5	3	8	6
wav2swf	2	3	2	2	3
total	**35**	**41**	**25**	**30**	**53**

From Table 2, we can observe that in nearly all these 10 metrics, Alphuzz discovers more bugs than the other AFL-based fuzzing techniques. For example, Alphuzz discovers 87 bugs through 10 rounds, with 4 more bugs than AFL and AFLFast, 11 more bugs than FairFuzz, and 14 more bugs than EcoFuzz. By examining the numbers of discovered bugs in Table 2, we observe that the performance of AFL is almost the same as AFLFast on CGC binaries. On the contrary, FairFuzz and EcoFuzz work worse than other fuzzing techniques on CGC binaries.

For AFL++-based fuzzing techniques, in nearly all these 10 metrics, Alphuzz++ discovers more bugs than AFL++ and AFL++-HIER. Alphuzz++ discovers 88 bugs through 10 rounds, with 4 more bugs than AFL++ and 3 more bugs than AFL++-HIER.

We present the detailed results of all the vulnerabilities discovered by these techniques in Appendix A.

5.4.2 *Unique bugs on UniFuzz*.

We first calculate the number of unique bugs discovered by each fuzzing technique. According to UniFuzz [25], we leverage the report produced by ASAN [37] and GDB [19] to de-duplicate bugs.

Table 3 shows the numbers of detected unique bugs by AFL-based fuzzing techniques. We can observe that Alphuzz finds no less unique bugs than other fuzzing techniques on 11 out of the

Table 4: Number of discovered unique exploitable bugs.

Binary	AFL	AFLFast	EcoFuzz	FairFuzz	Alphuzz
cflow	0	0	0	0	0
exiv2	0	1	1	1	1
flvmeta	1	1	1	1	1
gdk	4	3	2	0	3
imginfo	0	0	0	0	0
infotocap	0	0	0	0	0
jhead	0	0	2	0	0
jq	0	0	0	0	0
lame	2	1	1	1	2
mp3gain	1	1	0	0	1
mp42aac	0	0	0	0	0
mujs	0	0	0	0	0
nm	0	0	0	0	0
objdump	2	2	0	1	2
pdftotext	0	1	0	0	3
tcpdump	0	0	0	0	0
tiffsplit	1	1	1	1	1
wav2swf	2	3	2	2	3
total	**13**	**14**	**10**	**6**	**17**

Table 5: Number of discovered unique bug and exploitable vulnerabilities found by AFL++-based techniques.

Binary	Unique bugs			Exploitable bugs		
	AFL++	AFL++-HIER	Alphuzz++	AFL++	AFL++-HIER	Alphuzz++
cflow	3	5	5	1	1	1
exiv2	2	2	2	1	1	1
flvmeta	4	3	4	1	1	1
gdk	10	9	12	6	3	8
imginfo	0	1	1	0	0	0
infotocap	2	0	3	0	0	0
jhead	3	3	3	0	0	0
jq	0	0	0	0	0	0
lame	2	3	3	2	2	2
mp3gain	6	6	6	1	1	1
mp42aac	2	2	2	0	0	0
mujs	0	0	1	0	0	0
nm	0	0	0	0	0	0
objdump	2	3	3	2	1	2
pdftotext	2	8	4	2	2	3
tcpdump	0	0	1	0	0	0
tiffsplit	3	6	5	1	1	1
wav2swf	3	2	4	3	2	4
total	**44**	**55**	**59**	**20**	**15**	**24**

15 binaries. In total, Alphuzz finds 18, 12, 28, and 23 more unique bugs than AFL, AFLFast, EcoFuzz, and FairFuzz, respectively.

Table 5 shows the numbers of detected unique bugs by AFL++-based fuzzing techniques. We can observe that Alphuzz++ finds no less unique bugs than AFL++ on 14 binaries, and no less unique bugs than AFL++-HIER on 12 binaries. In total, Alphuzz++ finds 15 and 4 more bugs than AFL++ and AFL++-HIER, respectively.

5.4.3 *Exploitable vulnerabilities on UniFuzz.*

Exploitability reflects the severity of the vulnerability [25]. We leverage the GDB Exploitable [16] to identify exploitable vulnerabilities. GDB Exploitable classifies crashes into four categories: EXPLOITABLE, PROBABLY_EXPLOITABLE, PROBABLY_NOT_EXPLOITABLE, and UNKNOWN. Table 4 and Table 5 show the number of crashes that are classified as EXPLOITABLE and PROBABLY_EXPLOITABLE. We can observe that Alphuzz finds 4, 3, 7, and 11 more exploitable vulnerabilities than AFL, AFLFast, EcoFuzz, and FairFuzz, respectively. Alphuzz++ finds 4 and 9 more exploitable vulnerabilities than AFL++ and AFL++-HIER, respectively.

5.5 Code coverage

Code coverage is a critical metric for evaluating the performance of a fuzzing technique [43]. Basically, the more code a fuzzing technique can cover, the more likely it is to find the hidden bugs. According to previous studies [24, 35, 49], we use the bitmap maintained by AFL to measure the code coverage. Specifically, AFL maps each branch transition into an entry of the bitmap via hashing. If a branch transition is explored, the corresponding entry in the bitmap will be filled and the size of the bitmap will increase. As AFL provides real-time bitmap sizes, we can evaluate the increasing code coverage over time during the fuzzing process.

Table 6: p values of the code coverage on UniFuzz with Alphuzz as the baseline.

Binary	Alphuzz	AFL	AFLFast	EcoFuzz	FairFuzz
	AVG cov	p value	p value	p value	p value
cflow	4.859%	***0.0209***	***0.0408***	0.4795	***0.0078***
exiv2	15.728%	***0.0047***	***0.0023***	0.0511	***0.0054***
flvmeta	2.33 %	>0.10000	***0.0001***	0.0867	>0.9999
gdk	9.05%	***<0.0001***	***0.0014***	***0.0014***	***0.0014***
imginfo	8.324%	***0.0002***	***0.0047***	***0.0082***	***0.0002***
infotocap	5.267%	0.4221	0.3046	***<0.0001***	***0.0027***
jhead	1.477%	>0.9999	0.0698	***0.0031***	***0.0031***
jq	8.34%	***0.0435***	0.8951	***0.022***	0.1898
lame	14.161%	0.4648	0.2387	0.1008	0.4692
mp3gain	3.891%	0.9887	0.4027	0.1342	0.1013
mp42aac	7.734%	***0.0002***	***<0.0001***	***<0.0001***	***<0.0001***
mujs	12.984%	0.3047	***<0.0001***	***<0.0001***	***<0.0001***
nm	6.064%	0.1172	***0.0209***	***<0.0001***	***0.0217***
objdump	11.654%	0.8978	0.3047	0.4699	0.4695
pdftotext	25.904%	0.6415	***0.0024***	0.3035	0.9126
tcpdump	22.204%	***0.0009***	***<0.0001***	***<0.0001***	***<0.0001***
tiffsplit	5.231%	0.072	0.3227	0.7811	***0.0002***
wav2swf	1.39%	0.3067	***0.0086***	>0.9999	0.3043

We present the average bitmap size of the 10-round experiments of AFL-based fuzzing techniques for every binary in UniFuzz dataset. Figure 5 shows that Alphuzz can achieve higher or the same bitmap size than other techniques on 14 out of 18 binaries. For *mujs*, Alphuzz works worse than FairFuzz. For *nm*, Alphuzz works worse than EcoFuzz, AFLFast, and AFL. For *pdftotext*, Alphuzz works worse than AFLFast. For *tiffsplite*, Alphuzz works worse than AFL and FairFuzz.

By comparing the performance of Alphuzz with every baseline technique as shown in Figure 5, we can also observe that Alphuzz outperforms AFL, AFLFast, EcoFuzz, and FairFuzz on 15, 15, 16, and 14 binaries, respectively.

Table 7 shows the averaged bitmap size of AFL++-based fuzzing techniques. As shown in Table 7, Alphuzz++ achieves higher or the same bitmap size than AFL++ on 14 out of 18 binaries, AFL++-HIER on 15 out of 18 binaries.

5.6 Statistical Significance

To quantify whether there are significant differences between Alphuzz and other techniques, we leverage the Mann-Whitney U test to calculate the p value [21, 33] using Alphuzz as the baseline. According to Mann-Whitney U test [33], a p value less than 0.05 indicates a significant difference between two fuzzers.

As shown in Table 6, column 2 shows the average code coverage of Alphuzz. Columns 3-6 show the p values calculated by taking the code coverage of Alphuzz as the baseline. It indicates that Alphuzz significantly outperforms the baseline fuzzer, if p value is less than

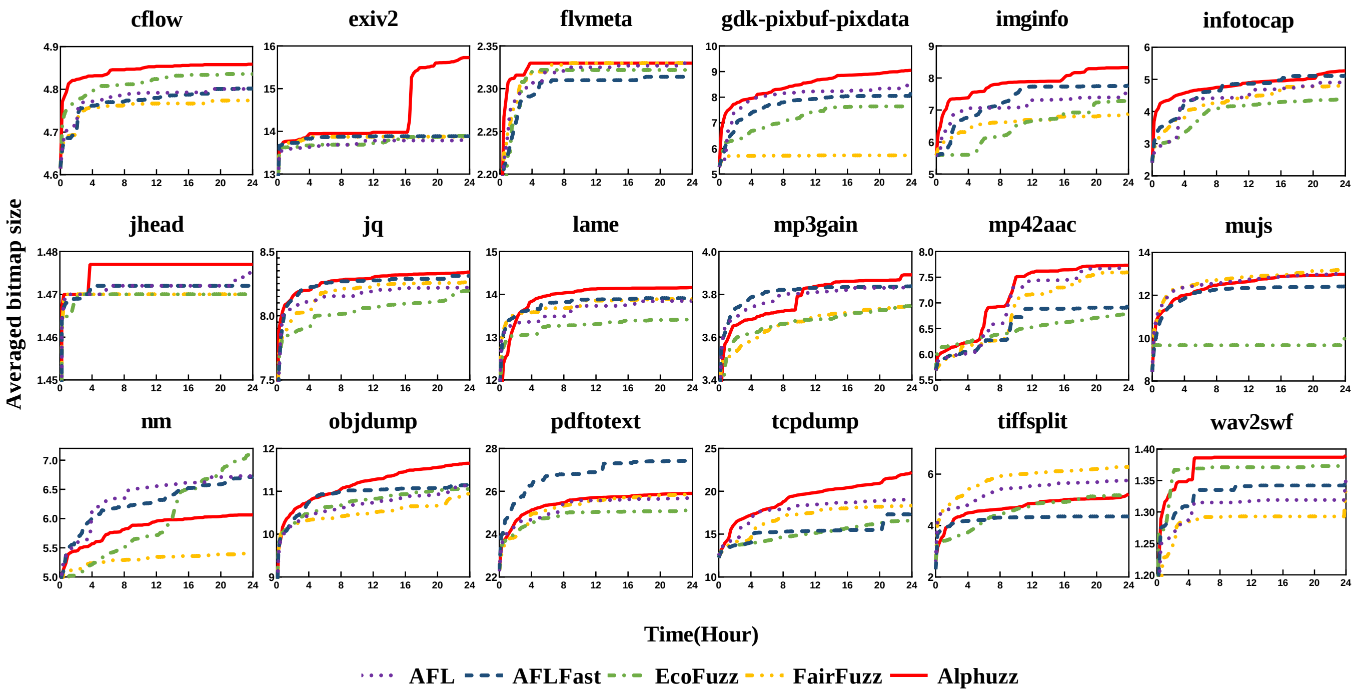

Figure 5: Average bitmap size for every binary of UniFuzz. Each graph represents a binary, with the abscissa representing the time and the ordinate representing the size of the averaged bitmap size.

Table 7: Averaged bitmap size and p values on UniFuzz with Alphuzz++ as the baseline.

Binary	Alphuzz++	AFL++		AFL++-HIER	
	AVG cov	AVG cov	p value	AVG cov	p value
cflow	***4.91%***	4.69%	***0.0008***	4.37%	***<0.0001***
exiv2	***28.85%***	24.23%	***0.0008***	25.00%	0.0635
flvmeta	2.15%	2.15%	>0.9999	2.15%	>0.9999
gdk	***10.44%***	10.19%	***0.00476***	9.26%	***0.0016***
imginfo	8.64%	8.65%	0.3155	8.70%	***0.0476***
infotocap	***5.97%***	5.78%	***0.0079***	2.64%	***<0.0001***
jhead	1.39%	1.39%	>0.9999	1.39%	>0.9999
jq	***8.51%***	7.60%	***0.0008***	7.68%	***0.0079***
lame	***14.57%***	14.33%	***0.0079***	14.01%	***0.0023***
mp3gain	***4.13%***	3.78%	***0.0008***	3.90%	0.3810
mp42aac	7.37%	6.89%	***0.0054***	7.66%	0.5238
mujs	***13.57%***	13.55%	***0.0412***	13.45%	***<0.0001***
nm	7.56%	7.27%	***0.0079***	7.56%	***0.0016***
objdump	***15.28%***	13.57%	***<0.0001***	13.82%	***<0.0001***
pdftotext	28.13%	28.22%	0.5004	28.45%	***0.0031***
tcpdump	***30.41%***	27.66%	***0.0008***	30.36%	0.6962
tiffsplit	6.20%	6.77%	0.0519	5.77%	***<0.0001***
wav2swf	1.29%	1.29%	>0.9999	0.87%	***<0.0008***

0.05 and the code coverage of Alphuzz is higher than the baseline fuzzer. We can observe that Alphuzz significantly outperforms AFL, AFLFast, EcoFuzz, and FairFuzz on 7, 9, 8, and 10 binaries, respectively.

Similarly, Table 7 shows the p values calculated by taking the code coverage of Alphuzz++ as the baseline. We can observe that Alphuzz++ significantly outperforms AFL++ on 12 out of 18 binaries, and outperforms AFL++-HIER on 9 out of 18 binaries.

5.7 Fuzzing throughput

Our seed scheduling strategy introduces both positive and negative impacts on the fuzzing throughput. On one hand, the tree structure can improve the fuzzing throughput in terms of time complexity. The searching time complexity of a tree structure is $O(log(N))$, whereas baseline fuzzing techniques organize the seed inputs in a queue, and the searching time complexity is $O(N)$. On the other hand, the implementation of our strategy requires collecting extra information, which may bring negative impacts on fuzzing throughput. To investigate the overhead of our strategy, we compare the average number of executions per second by AFL++, Alphuzz++, and AFL++-HIER on the UniFuzz dataset.

As shown in Figure 6, Alphuzz++'s throughput is no less than AFL on 12 out of 18 programs. That is, Alphuzz++ has a competitive throughput as AFL++. In addition, Alphuzz++ has higher executions per second than AFL++-HIER on 11 out of 18 programs. The result verifies our analysis that AFL++-HIER requires maintaining multiple coverage metrics, which reduces the fuzzing's throughput.

5.8 Impact of the parameter k

The parameter k in **Formula 2** is a constant, which controls the trade-off between exploration and exploitation. To investigate the impact of different k values on seed scheduling, we assign k as five different values 0, 0.014, 0.14, 1.4, and 14, respectively. Then, we conduct experiments on two datasets, the CGC dataset and the UniFuzz dataset.

Evaluation results show that the impact of k varies a lot for different datasets. Specifically, as shown in Table 8, Alphuzz achieves the highest code coverage with k as 1.4 than other configurations

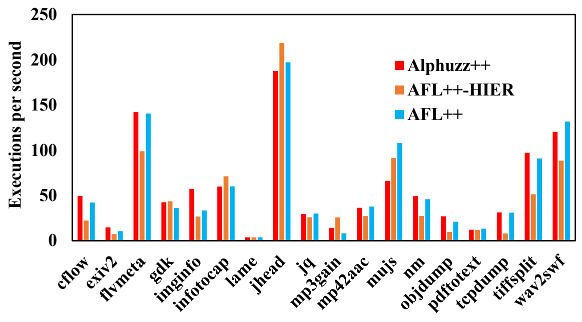

Figure 6: Throughput comparison.

Table 8: The Averaged edge coverage on CGC and UniFuzz with different values of the parameter k.

Dataset	Value of k				
	0	0.014	0.14	1.4	14
CGC	2.59%	2.61%	2.65%	2.75%	2.58%
UniFuzz	8.38%	8.46%	8.38%	8.23%	8.22%

on the CGC dataset. However, on the UniFuzz dataset, Alphuzz achieves the highest code coverage when k is set to 0.014.

For in-depth analysis, we further examine the fuzzing processes on different datasets. Our manual analysis shows that the binaries in UniFuzz are larger in scale and more complex than CGC binaries. Meanwhile, according to **Formula 2**, the value of *seed_score* for a seed will decrease with the increasing number of selected times. Thus, a smaller value of k will contribute to exploitation, which leads "seed mutation tree" to expand in the depth direction. Therefore, we can infer that k should be set as a smaller value with the increasing scale and more complex program structure.

5.9 Vulnerability detection on real-world binaries.

To assess whether Alphuzz can find vulnerabilities in real-world programs, we run the fuzzing techniques on 12 real-world binaries listed in Table 1. We collect the crashes discovered by each fuzzing technique, and then leverage *AddressSanitizer* [41] and *GDB* to distinguish redundant crashes and identify unique vulnerabilities.

Table 9 shows that these fuzzing techniques totally discover 12 vulnerabilities. Alphuzz discovers more vulnerabilities than other baseline techniques. We report the 12 vulnerabilities to upstream vendors. Among these vulnerabilities, 7 of them are confirmed by the vendors with the CVE-ID, 4 of them are confirmed but without a CVE-ID, and 1 of them has not been confirmed yet. Notably, we discovered 3 new vulnerabilities and obtained 3 **NEW CVEs**, which are **CVE-2021-25792**, **CVE-2021-25793**, and **CVE-2021-25794**.

Table 9: Vulnerabilities discovered in real-world binaries.

Binary	Vulnerabilities	AFL	AFLFast	EcoFuzz	FairFuzz	Alphuzz
cjpeg	CVE-2018-11214	✓	✓	✓	✓	✓
	CVE-2018-11212	✓	✓	✓	✓	✓
	issue*#5*	✗	✗	✗	✗	✓
infotocap	CVE-2018-19211	✗	✓	✓	✗	✓
	CVE-2021-25794	✗	✗	✗	✗	✓
mp3gain	CVE-2018-10778	✓	✓	✓	✓	✓
	CVE-2018-10777	✓	✓	✓	✓	✓
	CVE-2017-14406	✗	✗	✗	✓	✗
	CVE-2018-10776	✗	✗	✗	✗	✓
pdfimages	issue*#42073*	✗	✗	✗	✗	✓
	CVE-2021-25792	✗	✗	✗	✗	✓
	CVE-2021-25793	✗	✓	✗	✗	✓
Total	**12**	**4**	**6**	**5**	**5**	**11**

6 RELATED WORK

Fuzzing. In this paper, we mainly focus on seed scheduling for Coverage-based greybox fuzzing (CGF). Lots of fuzzing techniques leverage additional program analysis to improve the performance of fuzzing. Honggfuzz [40] and libfuzzer [36] introduce a data flow feature—the degree of matching of the operands of branch statements, and prioritized the selection of seeds that more satisfies the branch constraints. GreyOne [17] uses data flow analysis to determine the relationship between input fields and constraint-related variables, then schedules the input with the highest number of relevant key fields. MEUZZ [9] leverages sanitizer to measure how likely bugs can be triggered, and proposes an ML-based seed selection strategy for hybrid fuzzing.

Besides the seed scheduling strategies, there are a lot of fuzzing techniques focusing on different stages of fuzzing. HashFuzz [29] leverages universal hashing to increase the diversity of input values on which the executions traverse the same branch. MOPT [28] proposes a seed mutation strategy to determine the proper distribution for mutation operators. Entropy [5] develops a power schedule strategy, which gives more energy to seeds revealing more information about the program behaviors. NEUZZ [39] identifies the significance of program smoothing and uses an incremental learning technique to guide the mutation of fuzzing. Untracer [30] removes unnecessary instrumentation in basic blocks that have been explored to reduce the overhead of fuzzing.

Monte Carlo Tree Search. Monte Carlo Tree Search (MCTS) is a promising online planning approach, which has achieved great success in Go [38] and has had a profound impact on the field of artificial intelligence [14, 32]. The application field of MCTS is very extensive, such as active object recognition [34], wildlife monitoring[20], environment exploration [11, 23], and planetary exploration [1]. MCTS has been proposed in many different forms [7], but currently, the most common one is the Upper-Confidence Bounds Applied to Trees (UCT) algorithm[22]. Therefore, we use the UCT algorithm in our model.

MCTS also attracts the attention of researchers who focus on fuzzing. AFL-HIER [44] proposes a multi-level coverage metric, and leverages the UCT algorithm to perform seed selection. Legion [27] leverages MCTS to maintain a balance between concolic execution and fuzzing, and improves the performance of hybrid fuzzing.

7 CONCLUSION

In this study, we make a key observation that the mutation relationships among seeds are valuable for seed scheduling. To investigate the seed mutation relationships, we design a "seed mutation tree" and further propose an MCTS-based seed scheduling strategy by modeling the seed scheduling problem as a Monte-Carlo Tree Search (MCTS) problem. Evaluation shows that our approach outperforms other seed scheduling strategies with higher code coverage and more discovered vulnerabilities.

REFERENCES

[1] Akash Arora, Robert Fitch, and Salah Sukkarieh. 2017. An approach to autonomous science by modeling geological knowledge in a Bayesian framework. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2017, Vancouver, BC, Canada, September 24-28, 2017*. IEEE, 3803–3810.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *NDSS*, Vol. 19. 1–15.

[3] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine learning* 47, 2-3 (2002), 235–256.

[4] Hendrik Baier and Michael Kaisers. 2020. Guiding Multiplayer MCTS by Focusing on Yourself. In *IEEE Conference on Games, CoG 2020, Osaka, Japan, August 24-27, 2020*. 550–557.

[5] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. 2020. 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* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 678–689.

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

[7] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4, 1 (2012), 1–43.

[8] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. 711–725.

[9] Yaohui Chen, Mansour Ahmadi, Reza Mirzazade Farkhani, Boyu Wang, and Long Lu. 2020. MEUZZ: Smart Seed Scheduling for Hybrid Fuzzing. *CoRR* abs/2002.08568 (2020).

[10] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, Rajiv Gupta and Todd C. Mowry (Eds.). ACM, 265–278.

[11] Micah Corah and Nathan Michael. 2017. Efficient Online Multi-robot Exploration via Distributed Sequential Greedy Assignment.. In *Robotics: Science and Systems*, Vol. 13.

[12] Aleksander Czechowski and Frans A. Oliehoek. 2020. Decentralized MCTS via Learned Teammate Models. *CoRR* abs/2003.08727 (2020).

[13] DARPA. 2017. Cyber Grand Challenge Challenge Repository. http://www.lungetech.com/cgc-corpus/.

[14] Simon Demediuk, Marco Tamassia, Xiaodong Li, and William L. Raffe. 2019. Challenging AI: Evaluating the Effect of MCTS-Driven Dynamic Difficulty Adjustment on Player Enjoyment. In *Proceedings of the Australasian Computer Science Week Multiconference, ACSW 2019, Sydney, NSW, Australia, January 29-31, 2019*. 43:1–43:7.

[15] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

[16] Jonathan Foote. 2018. The exploitable GDB plugin. https://github.com/jfoote/exploitable.

[17] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20). USENIX Association, Boston, MA.*

[18] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. 679–696.

[19] GDB. 2019. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/.

[20] Benjamin Hefferan, Oliver M Cliff, and Robert Fitch. 2016. Adversarial patrolling with reactive point processes. In *Proceedings of the ARAA Australasian Conference on Robotics and Automation (ARAA, 2016)*. 39–46.

[21] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. 2123–2138.

[22] Levente Kocsis, Csaba Szepesvári, and Jan Willemson. 2006. Improved monte-carlo search. *Univ. Tartu, Estonia, Tech. Rep* 1 (2006).

[23] Mikko Lauri and Risto Ritala. 2016. Planning for robotic exploration based on forward simulation. *Robotics and Autonomous Systems* 83 (2016), 15–31.

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

[25] 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. 2020. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. *CoRR* abs/2010.01785 (2020).

[26] An-Jen Liu, Ti-Rong Wu, I-Chen Wu, Hung Guei, and Ting-Han Wei. 2020. Strength Adjustment and Assessment for MCTS-Based Programs [Research Frontier]. *IEEE Comput. Intell. Mag.* 15, 3 (2020), 60–73.

[27] Dongge Liu, Gidon Ernst, Toby Murray, and Benjamin I. P. Rubinstein. 2020. Legion: Best-First Concolic Testing. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, New York, NY, USA, 54–65.

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

[29] Hector D. Menendez and David Clark. 2021. Hashing Fuzzing: Introducing Input Diversity to Improve Crash Detection. *IEEE Transactions on Software Engineering* (2021), 1–1.

[30] Stefan Nagy and Matthew Hicks. 2019. Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. 787–802.

[31] Trail of Bits. 2016. DARPA Challenges Sets for Linux, Windows, and macOS. Accessed on September 10th 2020. https://github.com/trailofbits/cb-multios.

[32] Abdessamed Ouessai, Mohammed Salem, and Antonio Miguel Mora. 2020. Parametric action pre-selection for MCTS in real-time strategy games. In *Proceedings of the VI Congreso de la Sociedad Española para las Ciencias del Videojuego, On-line, October 7-8, 2020*. 104–115.

[33] p value. 2022. p value. https://en.wikipedia.org/wiki/P-value.

[34] Timothy Patten, Wolfram Martens, and Robert Fitch. 2018. Monte Carlo planning for active object classification. *Autonomous Robots* 42, 2 (2018), 391–421.

[35] Mohit Rajpal, William Blum, and Rishabh Singh. 2017. Not all bytes are equal: Neural byte sieve for fuzzing. *CoRR* abs/1711.04596 (2017).

[36] Kostya Serebryany. 2015. libFuzzer–a library for coverage-guided fuzz testing. *LLVM project* (2015).

[37] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2017. Addresssanitizer. https://github.com/google/sanitizers/wiki.

[38] Eren Sezener and Peter Dayan. 2020. Static and Dynamic Values of Computation in MCTS. *CoRR* abs/2002.04335 (2020). https://arxiv.org/abs/2002.04335

[39] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. 803–817.

[40] Robert Swiecki. 2017. Honggfuzz: A general-purpose, easy-to-use fuzzer with interesting analysis options. *URl: https://github. com/google/honggfuzz (visited on 06/21/2017)* (2017).

[41] The Clang Team. 2017. Addresssanitizer. http://clang.llvm.org/docs/AddressSanitizer.html.

[42] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. 579–594.

[43] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019.*

[44] Jinghan Wang, Chengyu Song, and Heng Yin. 2021. Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society.

[45] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing New Operating Primitives to Improve Fuzzing Performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 2313–2328.

[46] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In *29th USENIX Security Symposium (USENIX Security 20)*.

[47] M. Zalewski. 2017. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[48] Gen Zhang, Xu Zhou, Yingqi Luo, Xugang Wu, and Erxue Min. 2018. PTfuzz: Guided Fuzzing With Processor Trace Feedback. *IEEE Access* 6 (2018), 37302–37313.

[49] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society.

A EXPERIMENTAL RESULTS ON CGC DATASET

Table 10 presents the details of vulnerabilities that are discovered by AFL-based fuzzing techniques for at least once across the 10-rounds experiments. We mark the binaries of which the vulnerabilities are detected only by Alphuzz. A total of 91 vulnerabilities are discovered, of which 67 are discovered by all the fuzzing techniques. Alphuzz misses 4 of them. By contrast, AFL, AFLFast and EcoFuzz miss 8, 8 and 18 of vulnerabilities ever discovered. We can observe that Alphuzz discovers 6 unique vulnerabilities (CROMU_00061, CROMU_00064, KPRCA_00017, KPRCA_00019, KPRCA_00049, and KPRCA_00087) that are never discovered by other fuzzing techniques.

Table 11 presents the details of vulnerabilities that are discovered by AFL++-based fuzzing techniques for at least once across the 10-rounds experiments. We mark the binaries of which the vulnerabilities are detected only by Alphuzz++. We can observe that Alphuzz++ discovers 5 unique vulnerabilities (CROMU_00061, KPRCA_00017, KPRCA_00019, KPRCA_00049, and KPRCA_00087) that are never discovered by other fuzzing techniques.

Table 10: Detailed Vulnerabilities discovered by AFL-based fuzzing techniques.

CGC programs	Alphuzz	AFL	AFLFast	EcoFuzz	FairFuzz	CGC programs	Alphuzz	AFL	AFLFast	EcoFuzz	FairFuzz
CROMU_00004	✓	✓	✓	✓	✓	CROMU_00006	✓	✓	✓	✓	✓
CROMU_00009	✓	✓	✓	✓	✓	CROMU_00014	✓	✓	✓	✓	✓
CROMU_00015	✓	✓	✓	✓	✓	CROMU_00016	✓	✓	✓	✓	✓
CROMU_00018	✓	✓	✓	✓	✓	CROMU_00021	✓	✓	✓	✓	✓
CROMU_00023	✓	✓	✓	✓	✓	CROMU_00025	✓	✓	✓	✓	✓
CROMU_00026	✓	✓	✓			CROMU_00027	✓	✓	✓	✓	✓
CROMU_00031	✓	✓	✓	✓	✓	CROMU_00034	✓	✓	✓	✓	✓
CROMU_00035	✓	✓	✓	✓	✓	CROMU_00036	✓	✓	✓	✓	✓
CROMU_00038	✓	✓	✓	✓	✓	CROMU_00039	✓	✓	✓	✓	✓
CROMU_00040	✓	✓	✓	✓	✓	CROMU_00042	✓	✓	✓	✓	✓
CROMU_00044	✓	✓	✓	✓	✓	CROMU_00046	✓	✓	✓	✓	✓
CROMU_00048	✓	✓	✓			CROMU_00055	✓	✓	✓	✓	✓
CROMU_00057	✓	✓	✓	✓	✓	CROMU_00058	✓	✓	✓	✓	✓
CROMU_00061	✓					**CROMU_00064**	✓				
CROMU_00066	✓	✓	✓	✓		CROMU_00072	✓	✓	✓	✓	✓
CROMU_00076	✓	✓	✓		✓	CROMU_00077	✓	✓	✓	✓	✓
CROMU_00078	✓	✓	✓	✓	✓	CROMU_00079	✓	✓	✓		✓
CROMU_00082	✓	✓	✓	✓	✓	CROMU_00094	✓	✓	✓	✓	✓
CROMU_00095	✓	✓	✓	✓	✓	CROMU_00096	✓	✓	✓	✓	✓
CROMU_00097					✓	CROMU_00098	✓	✓	✓		✓
KPRCA_00010	✓	✓	✓	✓	✓	KPRCA_00011	✓	✓	✓	✓	✓
KPRCA_00012	✓	✓	✓	✓	✓	KPRCA_00014	✓	✓	✓	✓	✓
KPRCA_00017	✓					**KPRCA_00019**	✓				
KPRCA_00020	✓	✓	✓	✓	✓	KPRCA_00021	✓	✓	✓	✓	✓
KPRCA_00022	✓	✓	✓	✓	✓	KPRCA_00028	✓	✓	✓	✓	✓
KPRCA_00032	✓	✓	✓	✓	✓	KPRCA_00033	✓	✓	✓	✓	✓
KPRCA_00035	✓	✓	✓	✓	✓	KPRCA_00036	✓	✓	✓	✓	✓
KPRCA_00038	✓	✓	✓	✓	✓	KPRCA_00043	✓	✓	✓	✓	✓
KPRCA_00047	✓	✓	✓	✓	✓	**KPRCA_00049**	✓				
KPRCA_00050	✓	✓	✓	✓	✓	KPRCA_00051	✓	✓	✓	✓	✓
KPRCA_00052		✓	✓			KPRCA_00065	✓	✓	✓	✓	✓
KPRCA_00068				✓	✓	KPRCA_00069		✓	✓		
KPRCA_00081	✓	✓	✓			**KPRCA_00087**	✓				
KPRCA_00094	✓	✓	✓			KPRCA_00102	✓	✓	✓	✓	
KPRCA_00110	✓	✓	✓	✓	✓	NRFIN_00009	✓	✓	✓	✓	✓
NRFIN_00014	✓	✓	✓	✓	✓	NRFIN_00015	✓	✓	✓	✓	✓
NRFIN_00016	✓	✓	✓		✓	NRFIN_00018	✓	✓	✓	✓	✓
NRFIN_00021	✓	✓	✓	✓	✓	NRFIN_00022	✓			✓	
NRFIN_00023	✓	✓	✓	✓	✓	NRFIN_00024	✓	✓	✓	✓	✓
NRFIN_00026	✓	✓	✓	✓	✓	NRFIN_00035	✓	✓	✓	✓	✓
NRFIN_00038	✓	✓	✓	✓	✓	NRFIN_00039	✓	✓	✓	✓	✓
NRFIN_00040	✓	✓	✓	✓	✓	NRFIN_00041	✓	✓	✓	✓	✓
NRFIN_00042	✓	✓	✓	✓	✓	NRFIN_00049	✓	✓	✓		✓
NRFIN_00052	✓	✓	✓	✓	✓	NRFIN_00054		✓	✓	✓	✓
NRFIN_00061	✓	✓	✓	✓		NRFIN_00064	✓	✓	✓		✓
YAN01_00007	✓	✓	✓	✓	✓	YAN01_00012	✓	✓	✓	✓	✓
In Total	**87**	**83**	**83**	**73**	**76**						

Table 11: Detailed Vulnerabilities discovered by AFL++-based fuzzing techniques.

CGC programs	Alphuzz++	AFL++	AFL++-HIER	CGC programs	Alphuzz++	AFL++	AFL++-HIER
CROMU_00004	✓	✓	✓	CROMU_00006	✓	✓	✓
CROMU_00009	✓	✓	✓	CROMU_00014	✓	✓	✓
CROMU_00015	✓	✓	✓	CROMU_00016	✓	✓	✓
CROMU_00018	✓	✓	✓	CROMU_00021	✓	✓	✓
CROMU_00023	✓	✓	✓	CROMU_00025	✓	✓	✓
CROMU_00026	✓	✓	✓	CROMU_00027	✓	✓	✓
CROMU_00031	✓	✓	✓	CROMU_00034	✓	✓	✓
CROMU_00035	✓	✓	✓	CROMU_00036	✓	✓	✓
CROMU_00038	✓	✓	✓	CROMU_00039	✓	✓	✓
CROMU_00040	✓	✓	✓	CROMU_00042	✓	✓	✓
CROMU_00044	✓	✓	✓	CROMU_00046	✓	✓	✓
CROMU_00048	✓	✓	✓	CROMU_00055	✓	✓	✓
CROMU_00057	✓	✓	✓	CROMU_00058	✓	✓	✓
CROMU_00061	✓			CROMU_00064	✓	✓	✓
CROMU_00066	✓	✓	✓	CROMU_00072	✓	✓	✓
CROMU_00076	✓	✓	✓	CROMU_00077	✓	✓	✓
CROMU_00078	✓	✓	✓	CROMU_00079	✓	✓	✓
CROMU_00082	✓	✓	✓	CROMU_00094	✓	✓	✓
CROMU_00095	✓	✓	✓	CROMU_00096	✓	✓	✓
CROMU_00097	✓			CROMU_00098	✓	✓	✓
KPRCA_00010	✓	✓	✓	KPRCA_00011	✓	✓	✓
KPRCA_00012	✓	✓	✓	KPRCA_00014	✓	✓	✓
KPRCA_00017	✓			**KPRCA_00019**	✓		
KPRCA_00020	✓	✓	✓	KPRCA_00021	✓	✓	✓
KPRCA_00022	✓	✓	✓	KPRCA_00028	✓	✓	✓
KPRCA_00032	✓	✓	✓	KPRCA_00033	✓	✓	✓
KPRCA_00035	✓	✓	✓	KPRCA_00036	✓	✓	✓
KPRCA_00038	✓	✓	✓	KPRCA_00043	✓	✓	✓
KPRCA_00047	✓	✓	✓	**KPRCA_00049**	✓		
KPRCA_00050	✓	✓	✓	KPRCA_00051	✓	✓	✓
KPRCA_00052		✓	✓	KPRCA_00065	✓	✓	✓
KPRCA_00068			✓	KPRCA_00069		✓	✓
KPRCA_00081	✓	✓	✓	**KPRCA_00087**	✓		
KPRCA_00094	✓	✓	✓	KPRCA_00102	✓	✓	✓
KPRCA_00110	✓	✓	✓	NRFIN_00009	✓	✓	✓
NRFIN_00014	✓	✓	✓	NRFIN_00015	✓	✓	✓
NRFIN_00016	✓	✓	✓	NRFIN_00018	✓	✓	✓
NRFIN_00021	✓	✓	✓	NRFIN_00022	✓		
NRFIN_00023	✓	✓	✓	NRFIN_00024	✓	✓	✓
NRFIN_00026	✓	✓	✓	NRFIN_00035	✓	✓	✓
NRFIN_00038	✓	✓	✓	NRFIN_00039	✓	✓	✓
NRFIN_00040	✓	✓	✓	NRFIN_00041	✓	✓	✓
NRFIN_00042	✓	✓	✓	NRFIN_00049	✓	✓	✓
NRFIN_00052	✓	✓	✓	NRFIN_00054		✓	✓
NRFIN_00061	✓	✓	✓	NRFIN_00064	✓	✓	✓
YAN01_00007	✓	✓	✓	YAN01_00012	✓	✓	✓
In Total	**88**	**84**	**85**				