Balance Seed Scheduling via Monte Carlo Planning

Heqing Huang, Hung-Chun Chiu, Qingkai Shi, Peisen Yao, Charles Zhang

Abstract—Scheduling seeds, i.e., selecting a seed for mutation from a pool of candidates, significantly impacts the speed of a greybox fuzzer to achieve a target coverage rate. Despite much progress in improving seed scheduling, existing work cannot escape from the high-cost trap or the high-benefit trap: one line of approaches believe high cost implies high benefit and, thus, prefer the seeds that explore infrequently-visited paths; the other line of approaches directly calculate the potential benefits, e.g., the number of blocks able to cover, and prefer high-benefit seeds. Due to the ignorance of the impacts of either the cost or the benefits, they often trap fuzzers into mutating the seeds without increasing coverage. This paper presents BELIEFFUZZ, which transforms fuzzing into a Monte Carlo planning system with a upper confidence bound. The system allows us to dynamically compute both the benefits and the cost during the fuzzing process. The experimental results demonstrate that, to achieve the same coverage, our approach is significant more efficienct, with 2.12x-5.63x speedup and 1.18x-2.77x fewer executions needed, than the state of the art. Moreover, BELIEFFUZZ detected 31 more previously-unseen bugs in real-world projects, with 18 CVEs assigned.

Index Terms—Fuzzing, Seed Prioritization, Power Scheduling, Monte Carlo Planning.

1 INTRODUCTION

Greybox fuzzing has been proven to be one of the most powerful vulnerability detection methods [1], [2], [4], [7], [15], [25], [39]. With lightweight execution feedback, e.g., coverage, it preserves the test inputs that discover unseen program behaviors, e.g., new coverage, as seeds to further explore the target program. The seeds are prioritized and mutated to generate new test inputs. One critical challenge in this fuzzing procedure is finding an ideal prioritization of these seeds so that we can achieve a high coverage rate as soon as possible. To this end, existing efforts define many metrics to calculate the potential of new coverage for each seed. While these metrics have allowed for much improvement, we observe that they do not well-balance the *cost* and the *benefits* of scheduling a seed, thus easily trapping a fuzzer into ineffective seed mutation.

Many existing works focus on mutating high-cost seeds, e.g., assign higher potential to the seeds that explore paths with lower execution frequency [7], [37], [39], [40], [41]. Their key observation is that the infrequently visited paths have not been fully explored. Thus, mutating seeds of these paths may find new program behaviors. However, the low execution frequency of a covered path does not mean that the path relates to uncovered code. Keeping mutating these seeds may let a fuzzer stick at uninteresting paths without any coverage achievement, which we refer to as the *high-cost trap*. For example, in Figure 1, even though Seed 1 explores a path with the lowest execution frequency, we may assign higher potential to Seed 3 and Seed 4, because, compared to Seed 1, they execute two paths much "closer" to the uncovered blocks.[1]

• *H.Huang, H.Chiu, Q.Shi, P.Yao, C.Zhang are with the Department of Computer Engineering, The Hong Kong University of Science and Technology, China.*
E-mail:{hhuangaz, hchiuab, qshiaa, pyao, charlesz}@cse.ust.hk

1. Closeness represents the potential to reach the specific program points measured by a distance metric [6], [10], [25].

On the contrary, some other methods prefer mutating high-benefit seeds that are close to the uncovered blocks (e.g., Seed 3 and Seed 4 in Figure 1) [25], [33]. However, being closer to uncovered blocks does not mean it is easy to cover them. Thus, these approaches could let the fuzzer stick at ultra-complex path conditions without finding much coverage, which we refer to as the *high-benefit trap*. In the example of Figure 1, since Seed 3 is closer to more uncovered blocks (H, K, and L) than Seed 4, the fuzzer then focuses on mutating Seed 3 and expects to cover these blocks. However, the uncovered blocks, H, K, and L, are too difficult to cover with an extremely small probability. Thus, keeping mutating Seed 3 will consume many resources with few chances to improve the coverage.

To avoid falling into such high-cost or high-benefit traps, we advocate a new holistic model for greybox fuzzing to balance the cost and the benefits of scheduling a seed. For example, in Figure 1, we not only focus on mutating the most promising seed, Seed 3, the path of which dominates most uncovered blocks in the program, but also record the cost of mutation conducted toward this path. If no coverage is found, but unaffordable efforts have been made, even though it has the highest potential benefit, we temporarily give up this seed and switch to others, e.g., Seed 4. While the example is simple, a key challenge to resolve is to efficiently quantify the cost and the benefits of mutating a seed, which are not static but frequently change when the fuzzing procedure proceeds, the number of seeds increases, and the blocks are gradually covered.

Our key insight in addressing the challenge is that balancing the cost and the benefits of scheduling a seed can be solved by a Monte Carlo planning system with upper confidence bound (MCUCB) [19]. MCUCB can provide an optimal decision with a theoretical guarantee for a large search process. Even though many existing approaches borrow only a few concepts from the conventional Monte Carlo method [28], [37], [41], the adapted sampled probability for approximating the potential of each seed is based on the observed statistic, i.e., execution frequency, which cannot

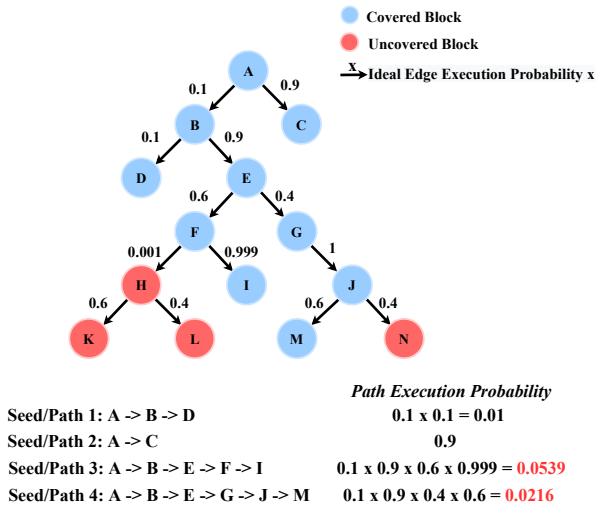


Figure 1: Motivating example. The figure illustrates a control flow graph with covered and uncovered basic blocks. The edge labels denote the probability of taking a branch. We assume that we have obtained four seeds (Seed 1 to Seed 4) during the fuzzing procedure.

model the benefits well. Instead, we leverage the strength of MCUCB, which allows the potential of intermediate states defined by the application scenario, to quantify both the cost and the benefit for the seeds in fuzzing. Therefore, seed prioritization considers the cost and the benefits simultaneously with a bounded coverage loss brought by not selecting the ideal seed. Specifically, we adopt the intuition from the cost-benefit analysis [22] to define the concept, "*confidence*", in MCUCB to dynamically measure whether this estimated potential benefit is still worth exploring with a confidence interval based on its cost. Furthermore, to improve the ineffectiveness caused by imperfect information, we arm our approach with a regret-based power scheduling method to early stop mutating the seeds that cannot bring new coverage as expected for other promising ones.

We have implemented our method as a greybox fuzzer named BELIEFFUZZ on top of AFL [1] and compared it to seven state-of-the-art coverage-based greybox fuzzers, AFL, AFL++ [15], FairFuzz [24], AFLFast [7], EcoFuzz [39], K-Scheduler [33], and Alphuzz [41]. The experimental results demonstrate a significant efficiency improvement, with 212%-563% speedup and 1.18x-2.77x fewer executions needed, over existing approaches to achieve the same coverage in a 24-hour experiment. Such efficiency improvement also benefits the code coverage, which is 117%-255% higher than the baselines. Moreover, BELIEFFUZZ detected 31 previously unseen bugs in real-world projects with 18 CVEs assigned. BELIEFFUZZ is publicly available.[2] Overall, this paper makes three contributions:

- We propose a novel probabilistic model that elucidates the weaknesses of existing works, showing the potential loss compared to the optimal coverage.
- We propose a new seed prioritization mechanism to improve the coverage of greybox fuzzing with less time cost.
- We provide an effective power scheduling strategy to avoid excessive energy allocation for seeds that cannot achieve much coverage.
- We provide empirical evidence that BELIEFFUZZ is more efficient and effective than the state-of-the-art greybox fuzzers.

2. https://github.com/belieffuzz/belieffuzz_artifacts

2 BACKGROUND AND MOTIVATION

This section discusses the problems behind existing methods (Section 2.1), explains the challenges (Section 2.2), illustrates the potential of Monte Carlo planning (Section 2.3), and, finally, briefly demonstrates our approach (Section 2.4).

2.1 Existing Seed Scheduling Models

Seed scheduling aims to maximize the coverage outcomes of a fuzzer by determining the potential of each seed. Specifically, it needs to make two sub-decisions — *seed selection*, which chooses the most promising seed from a pool, and *power scheduling*, which decides the number of mutations over this seed. The most widely used greybox fuzzer, AFL [1], uses multiple metrics, e.g., the coverage discovery and the execution time, to quantify the potential of each seed. If an input produced by mutating a seed detects a new path, AFL doubles the number of mutations allocated for this seed. Even though these heuristics are lightweight and have been proved useful occasionally, many existing approaches have demonstrated their limitations and proposed further optimizations [7], [37], [39]. To explain why high cost/benefit traps happen in existing works, we first discuss the intuition behind them, which are put into two categories: cost-oriented and benefit-oriented approaches.

Cost-Oriented Models. A number of existing efforts use the execution frequency of seed to calculate a transition probability, which estimates the cost of discovering unknown seeds from the discovered ones. Mainly, there are three closely related models: Markov chain [17], multi-armed bandit [5] and Monte Carlo tree search [20].

The first one, proposed by AFLFast [7], regards the fuzzing procedure as a stochastic searching process in the Markov chain. A Markov chain is a stochastic process with a sequence of states, X. Furthermore, it first proposes the concept of transition probability, p_{ij}, that indicates the probability of switching state X_i to state X_j at time t. AFLFast defines the state space as the discovered paths and their immediate neighbors, which are the paths that can be directly found by mutating the existing seeds, and defines the transition probability p_{ij} as the probability of generating an input that executes the path j from the seed that exercises the path i. AFLFast then selects the seed with a higher expectation to detect a new state j and deduces the energy allocated as $1/p_{ij}$. However, it is impossible to calculate such probabilities in practice. As a substitute, AFLFast approximates the probability, p_{i*}, of transiting from the path i to any other paths using its path execution frequency, f_i, as

$$p_{i*} = \frac{c}{f_i}$$

where c is the score provided by the original AFL. This formula encourages the fuzzer to explore the paths with

less execution frequency, which may lead to excessive exploration of high-cost paths.

The second approach, EcoFuzz [39], transforms the fuzzing procedure into a multi-armed bandit problem. The multi-armed bandit problem aims to maximize the outcomes from various choices (arms) with the benefits found in a limited number of playouts. Specifically, EcoFuzz approximates the probability, p_{i*}, of transiting from the path i, whose discovered index is also used for indicating the increasing difficulties in finding new paths at later stages, to any new paths as

$$p_{i*} = 1 - \frac{f_i}{\sqrt{i}}$$

which, albeit different from AFLFast, still drives the fuzzer to explore a path with less execution frequency, thus having the same problem.

Furthermore, AFL-Hier [37] extends the multi-armed bandit model with more fine-grained metrics to distinguish the seeds. Their intuition is that the fine-grained metrics could make the fuzzer select seeds with better diversity to explore different parts of the program. Specifically, it clusters the seeds into a multi-level tree by a set of gradually sensitive coverage metrics, e.g., function, block, and edge. It further defines the *rareness*, which is inverse proportion to the execution frequency, f, in each metric, to measure the score of each path i (in abbreviation) as

$$Score(i) \propto \frac{1}{f},\ \frac{1}{SelectionRatio_i}$$

where f is the averaged frequency that occurred in path i measured by each metric, and the $SelectionRatio_i$ is the selection frequency of the seed i. Even though the fine-grained coverage metric is more precise than solely using edge coverage, its basic intuition still directs fuzzing toward those less frequently visited paths and eventually falls into the *high-cost trap*.

Alphuzz [41] proposes to leverage the third model, Monte Carlo tree search (MCTS), which derives from the convention Monte Carlo method [28]. It forms the seed pool as a tree based on the seed discovery relation and represents the seed with the tree node. However, it still uses the execution frequency to measure the importance of the seeds. Specifically, it prefers the path of seed, i, exercising the most branches, b_{unique}, that have been detected solely be the seed, i,

$$Score(i) \propto |b_{unique}|,\ \frac{1}{SelectionRatio_i}$$

where the $SelectionRatio_i$ is the selection frequency of seed i. Nonetheless, this adaption cannot escape from the *high-cost trap* since the potential benefit of a seed is still not taken into consideration.

To sum up, even though existing studies propose various inspiring models, they still rely on the execution frequency to estimate the potential of each seed, which cannot escape from the *high-cost trap*.

Benefit-Oriented Models. The other trend of existing work is to utilize the potential benefit of each seed. Their intuition is to measure the benefit of seed from its reachable but uncovered part of the code with different predefined metrics. For example, Savior [11] uses the number of reachable bugs marked by the sanitizers as the prioritization standard. K-Schedule [33] adapts the graph centrality to approximate the number of reachable blocks for each seed. Cerebro [25] proposes multiple factors such as the complexity of the possible achievable coverage, to measure the benefit of the seed S, by solving a multi-objective optimization problem:

$$\begin{aligned} \max \quad & \textit{Benefit}(S) \\ \text{s.t.} \quad & \textit{Benefit}(S) = \sum_{s \in S} \mathcal{O}(M_1, M_2, \ldots, M_k) \end{aligned}$$

where $\mathcal{O}(M_1, M_2, \ldots, M_k)$ is the objective function determined by various k metrics, M_i ($i \in 1, \ldots, k$). Specifically, Cerebro proposes several heuristics to concretize the objective functions, which can be divided into two categories. First, it relies on the metrics used in AFL to measure the potential of the executed path, which shares the same intuition with AFL, i.e., the paths with less execution time but higher coverage are preferred. Second, it proposes to evaluate the potential achievable coverage as the benefit of each seed, which is the number of reachable but uncovered blocks of its executed paths. Cerebro then prefers to choose the seed with a higher probability of covering more paths.

However, the models used in these benefit-oriented methods, whose quality is highly dependent on the robustness of the given objective functions, cannot obtain these precise objective functions for fuzzing, which is similar to the transition probability. Although Cerebro and K-Scheduler propose several metrics as an approximation, it is infeasible to list all of them with theoretical formulas, i.e., the exact cost for achieving the coverage, and thus make the selection fall into the high-benefit traps. Furthermore, even though we can unconstrainedly add more metrics with manual expertise, such an enormous number of objectives could increase the burden of solving the optimization problem.

2.2 Challenges of Balancing the Cost and Benefit

Even though existing methods have the potential to select the seed effectively, they still cannot escape from the high-cost and the high-benefit traps simultaneously, which makes fuzzer allocate excessive resources to those seeds that cannot achieve much coverage. Specifically, we need to address the following two challenges.

Challenge 1: Simultaneously Quantifying the Cost and the Benefits. On the one hand, the models used in the *cost-oriented* methods naturally cannot consider both factors at the same time. Their approximation forms for the seed's potential only take one factor as input. Meanwhile, the multi-armed bandit model cannot tackle the fuzzing scenario since the number of the select targets (seed) is not fixed, thus, violating the basic assumptions of its methodology. On the other hand, even though the models used in the *benefit-oriented* method can consider both factors, a proper objective function for quantifying them still needs manual expertise without a guarantee provided.

Even though existing efforts [37], [41] attempt to borrow the concepts in the Monte Carlo method, i.e., Monte Carlo tree search, they cannot escape from the high-cost/benefit traps. At the high-level conceptual view, the Monte Carlo tree search (MCTS) used in Alphuzz is designed solely

based on the observable statistic, i.e., execution frequency, to measure the potential. Conventional MCTS is proposed uniquely for gameplay scenarios with multiplayer. Therefore, its intuition is to make the best decision based on the observable statistics in the gameplay without any forecast. To escape from the high-cost trap in fuzzing, we cannot only use the observable statistic, execution frequency, as Alphuzz to measure the potential of the seeds. Moreover, we also need to measure the potential benefit, which is the unobservable statistic, to help the fuzzer escape the high-cost trap mentioned in the paper.

Challenge 2: Avoid Trapping into Ineffective Seed Scheduling. Ideally, a fuzzer selects a seed based on its probability of achieving new coverage. However, since it is impractical to obtain the exact probability, a fuzzer cannot always select an ideal seed that can achieve maximal coverage. Thus, existing efforts leverage either cost or benefit to approximate the probability with different models for seed scheduling. Nevertheless, none of them is aware of the non-ideal selections caused by this imperfect information and help the fuzzer make up for the potential coverage loss compared to the ideal selection. Thus, our model needs to tackle these non-ideal selections for seed scheduling whenever we find out the selected seed is not the ideal one.

2.3 Monte Carlo Planning

Monte Carlo planning (MCP) has the potential to model fuzzing as an evolutionary decision-making process with a theoretical guarantee [8], [34]. MCP aims to find the optimal selection for the domains that cannot obtain complete knowledge from the enormous search space in any scenario without restrictions, i.e., two-players scenario for MCTS, which makes it capable to transform the seed selection problem as a Monte Carlo planning process.

Workflow of MCP. Specifically, it uses a stateful search tree to preserve the previous search statistics for optimizing the decisions afterward. Each node represents an intermediate action of the following searching process. MCP then uses a *tree policy* to search incrementally based on the previous results. The *tree policy* consists of four steps which are also shown in Figure 2:

1) *Selection*: Select a current optimal node on the tree for further exploration.
2) *Simulation*: Simulate the searching process.
3) *Action Preservation*: Expand the search tree with a successor node of the chosen one.
4) *Policy Update*: Update the selection policy according to the feedback from the simulation.

Specifically, MCP first determines the most valuable node of the current tree, which is expected to achieve the maximal benefit. With the selected node, MCP simulates the searching process defined by the problem domains to provide the benefit of node chosen, which is independent of the performance of the search algorithm. The flexibility of the simulations allows MCP to consider multiple factors in various domains effectively. If the simulation result is interesting, MCP expands the tree by preserving the simulated node. Then, use the statistics derived from the simulations updated for optimizing the following tree policy. Finally, MCP starts a new round of selection process to find the most promising node in the new search tree.

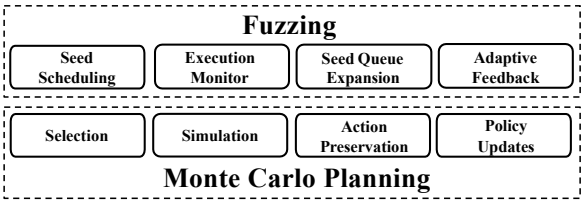


Figure 2: Workflow of the Monte Carlo planning process.

Analogy between Fuzzing and MCP. The similarities between fuzzing and MCP inspire us to leverage the concepts from MCP to solve the aforementioned problems in fuzzing. As shown in Figure 2, the initial seeds for fuzzing provide the initial states for examining the search space of the target program, which can be regarded as the root nodes of the search tree. The fuzzer then chooses the most promising seed to explore the target program in each iteration. Similarly, the tree policy first selects the node expected to bring the most benefit in the current tree. Then, mutating the selected seeds for new coverage can be regarded as a simulation for the current node. By monitoring the executions, the fuzzer enlarges its seed queues if new program behaviors are found, which can also become the condition to determine the expansion of the search tree. Finally, both fuzzing and MCP update the potentials based on execution feedback to maintain the effectiveness of selections in the following iterations.

Strength of MCP. As one of the state-of-the-art models for decision-making, MCP has two main strengths compared with other models: 1) *Generality for various domains*: One of the most significant strengths of MCP is the lack of need for domain-specific knowledge [35]. Specifically, the simulation stage in MCP is adaptable to various application domains by accepting any predefined reward function, which makes the decision-making process independent from the simulation. MCP can always provide an optimal solution with a predefined reward to update the selection policy regardless of the simulation stage. With the flexibility of the simulations in MCP, we can take both the cost and benefit of each seed when selecting the seed for various mutation strategies. Moreover, modeling fuzzing as an MCP process enables the optimized seed selection adapted to other advanced components in fuzzing besides seed scheduling, e.g., the choice of the mutation operators. 2) *Worse performance bound*: Another strength of MCP is that it is a state-of-the-art model providing minimal loss based on its selection strategy, which helps MCP achieve a better search result faster than other methods. The loss is the difference between the outcomes of the MCP and the one using the ideal selection with perfect information. Specifically, the performance of MCP takes the benefits from the following theorem.

Theorem 2.1 (Maximal Bias [19]). For K nodes selected for n times, $n > K$, the maximal bias of the achieved benefit compared with the ideal selection, which is represented by the expectation of the average *benefit* achieved from

each selection, $\mathbb{E}(\overline{benefit}_n)$, is bounded as

$$|\mathbb{E}(\overline{benefit}_n) - benefit^*| \leq \mathcal{O}(\frac{\ln n}{n}) \quad (1)$$

where $benefit^*$ is the *benefit* averaged over all ideal selections.

Thus, adapting the MCP model for fuzzing can optimize its seed selection with better effectiveness under guarantee.

2.4 Motivation of Using MCP

Based on the similarity between MCP and fuzzing, we follow its concepts of achieving its strength to solve challenges for balancing the cost and benefit in seed scheduling.

Solution 1: Quantifying the Cost and the Benefit with the Upper Confidence Bound. The generality of MCP enables us to design a proper potential considering the cost and benefit simultaneously to escape from the *high-cost/benefit traps*. In MCP, the selection policy aims to balance the exploration (check the node not examined yet) and exploitation (dig the node with a promising benefit) to maximize the outcomes [20]. This goal is equivalent to addressing the first challenge. Since MCP can estimate the potential of each node without knowing the perfect knowledge from expanding the whole search tree, we still need to design its benefit to make MCP consider the cost and benefit simultaneously while following the original theoretical bound. Specifically, it leverages the concept of *Upper Confidence Bound* (UCB) [19], as a confidence interval to quantify the closeness between the estimated benefit and the real one:

$$UCB_i = \bar{X}_i + \sqrt{\frac{2 \ln N}{n_i}} \quad (2)$$

where $\bar{X}_i$ is the achieved benefit for node i that can be defined by the problem domain knowledge, which allows us to consider cost and benefit simultaneously. n_i and N are the total number of selections of the node i and its parent, respectively. The constant $\sqrt{2}$ is proposed for making the overall potential converge within the computable bound of the Maximal Bias shown in Theorem 2.1. The flexibility of defining the $\bar{X}_i$ in MCP allows us to tackle the challenges by using both the cost and the benefit in UCB.

Solution 2: Measuring the Non-ideal Selection with Regret. The worse performance bound can serve as an indicator for fuzzing to avoid wasting efforts on those non-ideally selected seeds. To measure the worse performance bound, MCP utilizes the concept of *cumulative regret* [19] to measure the benefit loss due to not selecting the best seeds using incomplete knowledge, which is

$$regret = \sum_{i=0}^{n} |p^* - p_{n_i}| \quad (3)$$

where p^* is the ideal benefit achieved on average by the most promising node in each iteration, and p_{n_i} is the actual benefit achieved by the node, n, selected at round, i. Therefore, as this *cumulative regret* can indicate the loss brought by the non-ideal selection, minimizing the *regret* is equivalent to solving challenge 2. By measuring the *regret* of each selected seed, fuzzer is aware of the non-ideal selection and thus can further optimize the seed scheduling scheme. However, the ideal benefit, p^*, cannot be obtained in practice. Hence, we need to measure the *regret* with the proper metric from fuzzing.

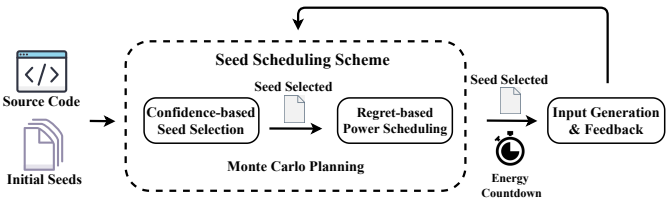


Figure 3: Adapting MCP for fuzzing in BELIEFFUZZ.

3 MODEL GREYBOX FUZZING AS MCUCB

In this section, we first illustrate the transformation from greybox fuzzing to the MCP system (Section 3.1). We then demonstrate how BELIEFFUZZ solves the first challenge by conducting seed selection using the defined UCB (Section 3.2). We then further tackle the second challenge by stopping the mutation before using up the assigned energy if it does not meet the expectation indicated by the MCP (Section 3.3).

3.1 Seed Discovery as Tree Expansion

BELIEFFUZZ regards fuzzing as an MCP process shown in Figure 2. The seeds which represent the intermediate searching states, are the vertices, $V : \{v_1, v_2, \ldots, v_k\}$, constructing the search tree, T, of the target program. The edge $\overline{v_i v_j}$ represents v_i discovers v_j. We regard the initial seed for fuzzing as the root vertex of the search tree. In each fuzzing iteration, we select the most promising seed in the current search tree. We consider the mutation cycles for the seed chosen in fuzzing as the simulation state. Afterward, we expand the search tree with the newly found seeds, $\{v'\}$. Finally, we update the execution statistics needed for selecting the seeds in the following iteration. Therefore, the seed selection problem then can be interpreted as which vertex should be chosen to achieve the maximal outcomes in each iteration.

Example 3.1. In Figure 4, we illustrate the transformed search tree of the program in Figure 1. The initial seed 1, which is the root node of the search tree, detects seeds 2, 3, and 4. We expand the tree by adding these new nodes as the successors of the root. Thus, the potential benefits of a node can be represented as the number of its successors. For instance, seed 3 can detect seeds 5 and 6, which respectively represent the paths $\overline{\text{ABEFHK}}$ and $\overline{\text{ABEFHL}}$. After the selected seed has been explored by fuzzing, its potentials are updated adaptively whenever a new seed is found.

The detailed workflow of BELIEFFUZZ is presented in Algorithm 1. We initialize the given seed for fuzzing at Line 3. In each fuzzing iteration, we select the most promising seed in the current search tree at Line 5 based on our UCB-based selection (Solution 1) at Line 12. Then, we leverage fuzzer to conduct mutation as simulation at Line 6. To minimize the potential coverage loss, BELIEFFUZZ uses regret-based scheduling for handling the non-ideal selections of the seed

Algorithm 1 Model fuzzing as MCUCB in BELIEFFUZZ.

```
1:  procedure FUZZING(S_init)
2:      S_init, initial seeds.
3:      T ← S_init                                   ▷ Initialization
4:      while Within Time Budget do
5:          v ← FindOne(T)                           ▷ Selection
6:          {v'} ← Fuzz(v, T)                        ▷ Simulation
7:          T ← T ∪ {v'}                             ▷ Expansion
8:          T ← Update(v, T)                         ▷ Back Propagation
9:      end while
10: end procedure

11: procedure FINDONE(T)
12:     return max_{v∈T} X̄_v + sqrt(ln N / n_v)      ▷ UCB-based Selection (§ 3.2)
13: end procedure

14: procedure FUZZ(v, T)
15:     P ← EnergyAlloc(v)
16:     return Mutation(v, P, T)                     ▷ Regret-based Scheduling (§ 3.3)
17: end procedure

18: procedure UPDATE(v, T)
19:     n_v ← n_v + 1
20:     for all v ∈ T do
21:         recompute X̄_v
22:     end for
23:     return T
24: end procedure
```

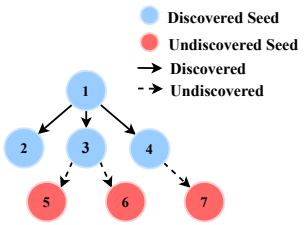


Figure 4: Transformed tree of the motivating example.

(Solution 2) at Line 16. Afterward, we expand the search tree with the newly found seeds, $\{v'\}$ at Line 7. Finally, we update the execution statistics needed for calculating the UCB at Line 12 in the following iteration at Line 8.

3.2 Quantify the Seed Potential with the Variant Upper Confidence Bound

To quantify the cost and benefit of each seed without modifying the original design of MCP (Challenge 1), BELIEFFUZZ adapts the concept of *benefit-cost ratio* in cost-benefit analysis [22] to define the UCB in the selection phase of MCP (Line 12 in Algorithm 1) according to its original design in Equation 2. Specifically, we quantify the benefit and costs to avoid falling into the traps as:

Definition 3.1 (Benefit). $\bar{X}_v$ in UCB is defined as the potential benefit of a seed, *Benefit*, which measures the potential coverage from mutating the seed v:

$$Benefit_v = \frac{r_v}{n_v} \tag{4}$$

where r_v is the number of the potential paths found from mutating seed v, n_v is the times of selecting seed v.

To properly quantify the *Benefit* for each seed mentioned in Equation 4, we need to measure the number of paths that can be discovered from each seed. Our basic idea is to accumulate the potential coverage of all the blocks covered by a seed. Specifically, we define the benefit, r_b, of a block, b, as the number of successor blocks in the inter-procedure control flow graph. For instance, there are three uncovered blocks from basic block F in Figure 1. Based on this block benefit, we further define the benefit of the seed v, r_v, as

$$r_v = \sum_{\forall b \in P_v} r_b \tag{5}$$

where P_v is the path executed by seed v.

However, such an accumulated benefit may compute the block redundantly, thus introducing a bias for the seed prioritization. For example, Figure 5 is a variant subprogram of the motivating example in Figure 1. Suppose that we accumulate the benefits from each block of its execution path. In that case, the priorities of seeds 1 and 2 are reversed since the redundantly accumulated benefits of seed 2 from blocks E, G, and J, which is 9, is larger than the one of seed 1, which is 8.

To overcome the above issue, our intuition comes from the original seed mechanism in fuzzing: The newly-preserved seed is used to detect the uncovered paths close to the newly-discovered program. For example, in Figure 1, the reason for preserving seed 3 is that it detects a new block F closer to the uncovered blocks H, K, and L, which are more likely to cover them compared with other seeds.

Based on the intuition, we only record the benefits using the number of the uncovered blocks, i.e., blocks F and I in Figure 1, from the control flow graph, CFG, whose back edges are removed. Specifically, we calculate the number of uncovered successors from all covered blocks, B, of each seed using the following steps:

1) $\forall b \in B, \forall E(b, b_i) \in$ CFG, if b_i is uncovered, $b_i \in S$
2) $\forall b \in S, \forall E(b, b_j) \in$ CFG, if b_j is uncovered, $b_j \in S_r$
3) if $(S_r \cup S) \neq S$, go back to step 2 with $S \leftarrow (S_r \cup S)$

where $E(b_i, b_j)$ is the edge between block b_i and b_j in CFG.

The size of the final deduplicated set S_r is the benefit of B. Using the three steps on the control flow graph, we can obtain the number of distinct uncovered blocks that are reachable for each seed. Therefore, the benefit of seeds 1 and 2 in Figure 5 are 3 and 2 in BELIEFFUZZ, which provides a more precise benefit measurement than the accumulation-based method. Meanwhile, BELIEFFUZZ adaptively normalizes the benefit by the maximal coverage achievable by a single seed in the current seed queue to satisfy the assumptions that the benefit should be in $[0, 1]$ in Theorem 2.1. Moreover, we dynamically update the *benefits* of each seed according to the coverage discovery. For example, if any uncovered block, i.e., block H, K, or L, has been covered during the fuzzing process, BELIEFFUZZ then updates the benefit of seeds reachable to this block.

The potential benefit drives BELIEFFUZZ to select the seed that can find more coverage defined by r_v. Moreover, BELIEFFUZZ gradually loses expectation for those potential benefits not achieved during the increase of selection, n, which avoids excessively selecting the seed towards the path conditions that are too complex to cover. Besides, instead of measuring the benefits based on the achieved coverage in existing work [25], [37], BELIEFFUZZ leverages

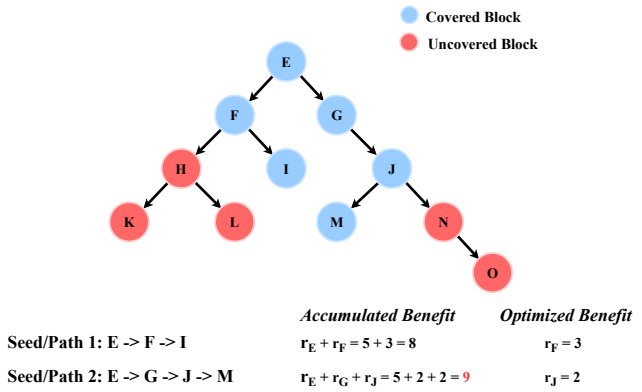


Figure 5: Comparison of accumulated benefit with optimized one used in BELIEFFUZZ.

the potential coverage to prevent fuzzing the well-explored program for better efficiency.

Definition 3.2 (Confidence). Each seed has a confidence, *Confidence*, which measures the possibility of achieving the benefit of the seed v by its cost ratio as

$$Confidence_v = \sqrt{\frac{2 \ln N}{n_v}} \tag{6}$$

where N is the times of selecting the parents of seed v.

Apart from prioritizing the seed with higher benefits, BELIEFFUZZ also places more confidence in the seed with fewer selections since the cost ratio increases gradually from the failures to detect the coverage. Unlike execution frequency, selection frequency follows the design of UCB while not involving frequent updates. Overall, we have the defined UCB for each seed, representing the expected possibility of finding new coverage by combining the two aspects:

$$UCB_v = Benefit_v + Confidence_v \tag{7}$$

Thus, the fuzzer selects the seed that achieves the highest coverage benefit while using the minimum mutation efforts measured by its costs.

Remark. First, we combine the achievable potential coverage as the benefit with selection frequency as the cost for each seed, which makes fuzzing avoid falling into high-cost/benefit traps. Second, we want to emphasize that unlike existing work [41] violates the design of their model, the MCUCB used in BELIEFFUZZ can maintain the effectiveness of the original models. Monte Carlo-based approach should select the node from any node that may explore new states [13]. Instead, Alphuzz completely ignores the intermediate seeds that can find new coverage in fuzzing, which hinders the effectiveness of examining the programs. Therefore, Alphuzz cannot maintain the theoretical bounds proposed in the Monte Carlo method as BeliefFuzz does. Instead, BELIEFFUZZ does not compromise the design of the Monte Carlo planning by regarding all the seeds as selectable. We have demonstrated the empirical evidence from the evaluation mentioned in Section 4.4.

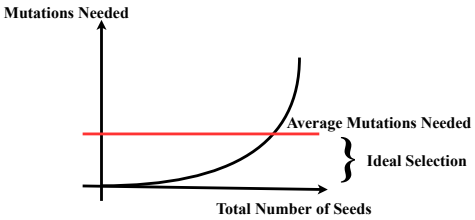


Figure 6: Intuition for regret-based power scheduling. The ideal-selected seed should find new coverage within the average expectation among all seeds.

3.3 Early Terminate the Mutation with Regret

Even though we use the UCB to select the seeds with a theoretical guarantee, the seed chosen may not be the most promising one due to incomplete information (Challenge 2). To further make up for the potential coverage loss caused by these non-ideal selections, our intuition is to terminate the mutation iteration of the selected seed earlier when it cannot meet the expectation to discover new paths (Line 16 of Algorithm 1). Specifically, we adapt the concept of *cumulative regret* in MCP for measuring the correctness of each selection to avoid excessively mutating the seeds that do not achieve the coverage as expected.

However, as we discussed in Section 2.4, it is impractical to obtain the ideal achievable coverage, p^*, to calculate the *regret* shown in Equation 3. To address this issue, we propose utilizing the number of mutations needed for finding new coverage as the metric to measure the *regret*. We regard p^* in Equation 3 as the minimum mutation that should use for finding new paths, and p_{v_i} is the actual mutation used for the seed chosen, v, at the round, i. Intuitively, no matter what evolutionary input generation method is used, the ideal selection should find the seed that can achieve the most coverage with the least number of mutations. With the transformed *cumulative regret*, we minimize the potential coverage loss using an optimized power scheduling mechanism by stopping the mutation that does not meet the minimal expectation of finding new coverage. More specifically, if the chosen seed has not found any new coverage using the expected number of mutations, this iteration should be stopped at once, even if it has not used up assigned energy.

To properly quantify this expectation, our basic intuition, illustrated in Figure 6, is that the chosen seed, which is expected as the most promising one in the seed queue, should need no more than the average number of mutations needed for achieving new coverage. Based on this intuition, BELIEFFUZZ records the number of mutations used to detect the first new path in each iteration of a selected seed as its *energy used*, and the average number of mutations used to detect all paths found. We then utilize these two metrics as the minimum expectation to stop the mutation iteration of the seeds chosen.

Algorithm 2 shows the detail of our design. Before mutating each selected seed, we first obtain the total number of mutations, M_{all}, from the tree, T, and the number of seeds found by the selected seed, F_v, with its *energy used*, E_v and mutation conducted, M_v of the seed chosen at Lines

Algorithm 2 Regret-based scheduling

```
1: procedure MUTATION(v, P, T)
2:     v, seed chosen. P, assigned energy. T, current tree.
3:     E_v, amount of energy used of v.
4:     F_v, number of seeds found by v.
5:     M_v, M_all, number of mutations done for v and all seeds.
6:     {v_new} ← ∅
7:     n_current, n_last, found ← 0
8:     while P ≠ 0 do
9:         if F_v > 0 then
10:            if |{v_new}| then
11:                Exp_energy ← M_v/F_v
12:            else
13:                Exp_energy ← E_v/F_v
14:            end if
15:        else
16:            Exp_energy ← M_all/|T|
17:        end if
18:        n_current ← n_current + 1
19:        if n_current − n_last > Exp_energy then
20:            end_mutation()
21:        end if
22:        found, v' ← fuzz_one(v)
23:        if found then
24:            E_v ← E_v + n_current − n_last
25:            F_v ← F_v + 1
26:            n_last ← n_current
27:            {v_new} ← {v_new} ∪ {v'}
28:        end if
29:        P ← P − 1
30:    end while
31:    return {v_new}
32: end procedure
```

3-5. There are three expectations that BELIEFFUZZ used to evaluate whether the selected seed is the ideal one.

- First, if the seed has not found any new path yet, BELIEFFUZZ expects this seed to discover the first new path within the average number of mutations, $M_{all}/|T|$, used for detecting a new path according to the overall seed at Line 16. If it has found at least one new path, BELIEFFUZZ updates the used energy to adjust this threshold dynamically for later mutation.
- Second, BELIEFFUZZ expects the chosen seed to discover new coverage within its average performance to prove the correctness of selection. Specifically for each seed selected, BELIEFFUZZ expects the chosen seed to cover at least one new path within its average *energy used* for finding new paths, E_v/F_v at Line 13.
- Third, BELIEFFUZZ anticipates finding new paths according to its average mutation needed, M_v/F_v at Line 11, after finding a new path in this iteration, which is more relaxed than the previous expectation since it has achieved the minimum goal of this iteration to find new coverage. Whenever the chosen seed cannot meet either of the expectations,

BELIEFFUZZ stops mutating it to avoid further energy waste at Lines 19-20. Finally, the discovered seeds, $\{v_{new}\}$, are preserved for expanding the search tree at Line 31. Therefore, BELIEFFUZZ can avoid wasting excessive efforts on those seeds without much coverage outcome. Noted that our method can be adapted to various mutation strategies: Even though different mutation strategies could influence the effectiveness of finding coverage, the average number of mutations needed for detecting new coverage is also changed, which allows us to adjust the regret dynamically.

Table 1: Baseline fuzzers.

ID	Fuzzer	Description
1	AFL [1]	Most widely-used evolutionary fuzzer
2	AFL++ [15]	AFL integrated with multiple advanced optimizations
3	FairFuzz [24]	Masked mutation for less-frequent branches
4	AFLFast [7]	Sophisticated power scheduling with Markov chain
5	EcoFuzz [39]	Multi-armed bandit model for seed selection
6	Alphuzz [41]	MCTS model solely using execution frequency
7	K-Scheduler [33]	Model seed's potential with graph centrality

Table 2: Real-world benchmark programs and vulnerabilities. The project size is measured in thousands of lines of code. T_r is the time costs for the offline reachability analysis providing the potential benefit for each branch.

ID	Project	Version	Input format	Size	T_r	Program
1 2	Bento4	1.6.0.0	MP4	100	6	mp42aac mp4info
3 4	Binutils	2.36	ELF	4722	53	nm-new objdump
5	Faust	2.34.6	DSP	452	67	faust
6	Fig2dev	3.2.8b	FIG	51	4	fig2dev
7	Gpac	1.0.1	MP4	1063	153	mp4box
8	ImageMagick	7.1.0-3	BMP	555	78	magick
9	Libjpeg-turbo	2.1.1	JPG	89	34	djpeg
10 11 12	Libtiff	4.3.0	TIFF	978	12	tiff2pdf tiff2ps tiffcrop

4 EVALUATION

We implemented BELIEFFUZZ, a greybox fuzzer with an optimized seed scheduling based on LLVM [21]. As shown in Figure 3, BELIEFFUZZ takes the program source code and initial seeds as its inputs. The input source code is compiled to LLVM bitcode, on which the static analysis that computes the initial benefit of each block is performed. By default, we use AFL as the fuzzing engine to select seeds and reschedule energy for mutation, allowing BELIEFFUZZ to be integrated with various AFL-based greybox fuzzing engines, such as Mopt [2] and FairFuzz [24].

4.1 Evaluation Setup

The evaluation consists of three parts. First, we compared BELIEFFUZZ against state-of-the-art fuzzers from two aspects, coverage and vulnerability discovery (Section 4.2). Second, to understand how BELIEFFUZZ selects and assigns proper energy for the seeds, we also evaluated how much our strategies contribute to reducing the time cost of fuzzing (Section 4.3). Third, we illustrate the strength of our method and discuss the potential future improvement (Section 4.5).

Baselines. We compared BELIEFFUZZ with the fuzzers shown in Table 1. Their technical details are mentioned in Section 2.1. AFL [1] is the most widely used greybox fuzzer. AFL++ [15] integrates AFL with multiple optimizations proposed by the state of the art. AFLFast [7], EcoFuzz [39], and AFL-Hier [37] are three recent greybox fuzzers that prioritize the most promising seeds to maximize the coverage in a given time budget using execution frequency. We noticed

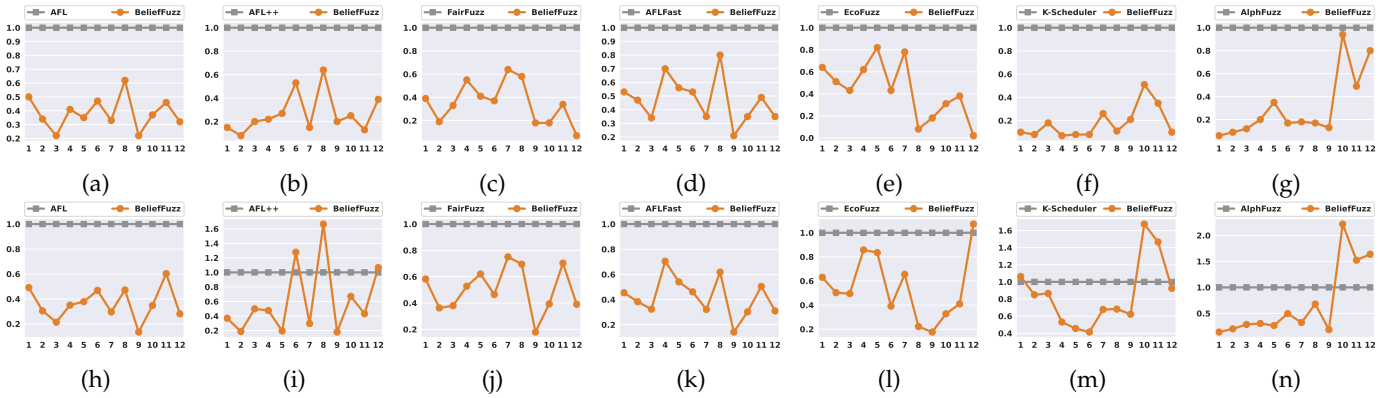


Figure 7: Comparison of achieving the same coverage conducted by each benchmark fuzzer, AFL, AFL++, FairFuzz, AFLFast, EcoFuzz, K-Scheduler, and Alphuzz in 24 hours using BELIEFFUZZ. Figure 7a) to 7g) present the amount of time needed for BELIEFFUZZ to achieve the coverage. Figure 7h) to 7n) present the number of executions needed for BELIEFFUZZ to achieve the coverage. The x-axis represents the programs identified using the index shown in Table 2. The y-axis denotes the improved ratios compared with the benchmark fuzzers. We use the performance of the benchmark fuzzer as the baseline to demonstrate the improvements brought by BELIEFFUZZ. The results are the average outcomes from running each fuzzer ten times.

that AFL-Hier has functionality issues when evaluating target programs and their results cannot be reproduced. Since its authors have not replied to our request, we do not compare it in our evaluation. Alphuzz [41] is one recent paper that uses MCTS but only relies on execution frequency for selection criteria. FairFuzz [24] prefers the rare-executed branch and adapts a musked mutation strategy to avoid mutating the related input bytes. K-Schedule [33] uses graph centrality to approximate the potential coverage that can be achieved by each seed. We make all chosen fuzzers and BELIEFFUZZ skip the deterministic stage of mutation to make a fair evaluation the effectiveness of the seed scheduling mechanism.

Benchmarks. We evaluated our tools on a few well-evaluated real-world programs to show the generality of the improvement. We also chose 12 binaries in 8 real-world programs that have been frequently evaluated in existing literature [7], [24], [25], [33], [39]. These programs, shown in Table 2, have diverse functionalities with various input formats. Meanwhile, we provide the time cost of the initial static analysis, T_r, for each project, all of which is less than 3 minutes. Such small time cost is acceptable compared to the 24-hour time budget for fuzzing.

Configurations. The initial seed corpus decides the effectiveness of fuzzing [31]. To achieve the best performance of related work, we used the seeds provided by AFL in their official GitHub repository[3]. The seeds for *faust*[4] and *fig2dev*[5] were provided by their GitHub repositories. We ran every experiment 10 times with a 24-hour time budget for each time. We also employed the Mann-Whitney U Test [27] to demonstrate the statistical significance of the contribution made by each part of our framework. All experiments were conducted on an Intel Xeon(R) server using Ubuntu 20.04 LTS with four Platinum 8358 CPUs and 512GB of memory.

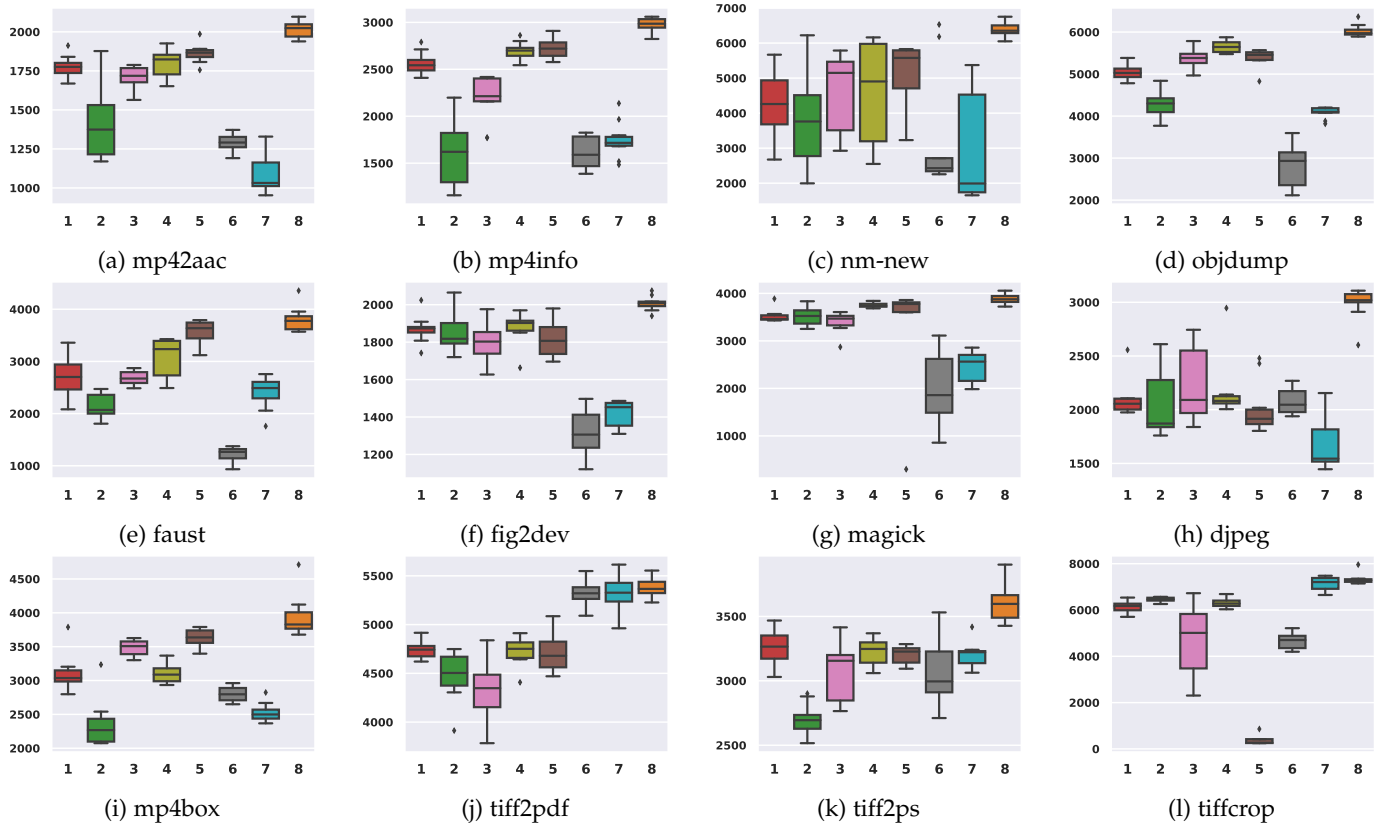
3. https://github.com/google/AFL/tree/master/testcases
4. https://github.com/grame-cncm/faust/tree/master-dev/tests
5. https://sourceforge.net/p/mcj/fig2dev/ci/master/tree/fig2dev/tests/data/

4.2 Comparing to the State of the Art

We first demonstrate the efficiency brought by our seed scheduling model. Specifically, we evaluated the time cost and the number of executions needed by BELIEFFUZZ to obtain the same coverage achieved by each baseline fuzzer in 24 hours on the benchmark programs. To avoid the coverage interference caused by the bitmap collision [16], we use the edge coverage metric in AFL++ [15], which follows the design of CollAFL to resolve the collision issue.

Figure 7 lists the comparison results. In general, BELIEFFUZZ outperforms existing work in covering the program more efficiently, with 3.22x speedup on average in all programs. Specifically, BELIEFFUZZ is 2.63x, 3.74x, 2.85x, 2.12x, 2.32x, 5.63x, and 3.24x faster on average compared to AFL, AFL++, FairFuzz, AFLFast, EcoFuzz, K-Scheduler, and Alphuzz to achieve the same coverage within 24 hours with all p-values less than 0.01. Furthermore, with the optimized seed selection and power scheduling model, BELIEFFUZZ only needs 2.77x, 1.64x, 2.00x, 2.38x, 1.82x, 1.18x, and 1.45x fewer executions for achieving the same coverage compared with the benchmark fuzzers, which is 1.89x fewer on average. Such a large proportion of time and execution reduction allows BELIEFFUZZ to save many computational resources. We notice that BELIEFFUZZ may execute more to find the same coverage compared with AFL++, EcoFuzz, K-Scheduler and Alphuzz in same project shown in Figures 7i, 7l, 7m and 7n. Nonetheless, BELIEFFUZZ can achieve it much faster with higher coverage overall, according to Figures 7b, 7e, 7f and 7g, which shows these fuzzers could stick at some paths with a long execution time. Moreover, we can also notice that Alphuzz does not show much improvement compared with others, which proves the impact of effectively balancing the cost and benefit simultaneously with MCUCB rather than the conventional MCTS.

Moreover, such efficiency improvement also enhances the effectiveness in terms of coverage. Figure 8 demonstrates the achieved coverage based on the previous experiment. On average, BELIEFFUZZ achieves 1.23x, 1.43x, 1.27x, 1.17x,

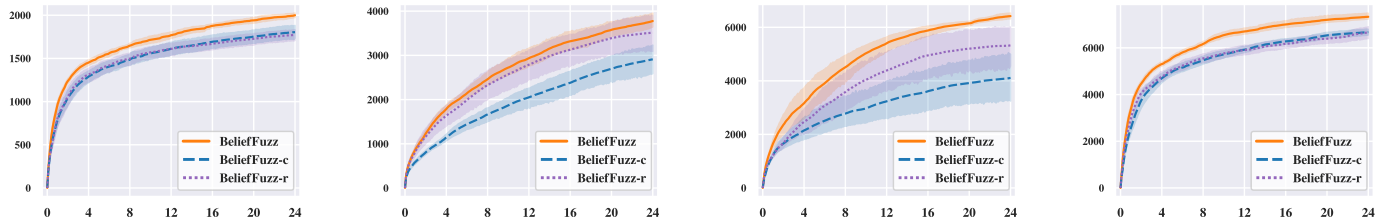


(a) mp42aac (b) mp4info (c) nm-new (d) objdump

(e) faust (f) fig2dev (g) magick (h) djpeg

(i) mp4box (j) tiff2pdf (k) tiff2ps (l) tiffcrop

Figure 8: The number of paths discovered by the different fuzzers in 24-hour experiments averaged over ten runs. The x-axis represents the fuzzers identified using the index shown in Table 1. The last entry represents the results of BELIEFFUZZ. The y-axis demonstrates the number of paths found according to the results in 10 runs.



(a) mp42aac (b) faust (c) nm-new (d) tiffcrop

Figure 9: Coverage achieved by BELIEFFUZZ, BELIEFFUZZ-*c*, and BELIEFFUZZ-*r* averaged in ten runs within 24 hours. We use the results of BELIEFFUZZ as the baseline. The x-axis is the time spent by fuzzers. The y-axis is the coverage achieved by different fuzzers. The statistics are averaged over ten runs, and the shadow denotes the 95% confidence interval.

2.55x, 1.72x, and 1.53x more coverage than AFL, AFL++, FairFuzz, AFLFast, EcoFuzz, K-Scheduler, and Alphuzz, respectively. The overall improvements represent that BELIEFFUZZ assigns proper energy for the higher-potential seed to detect new coverage with less resource wasted.

In the meantime, BELIEFFUZZ detects 47 bugs in the evaluation projects shown in Tables 3 and 4, which is 31 more than the baseline fuzzers on average. The details of these bugs can be found through this link.

4.3 Ablation Study

To further understand the contribution of the MCUCB model used in BELIEFFUZZ, we evaluated the contribution of the two individual components, namely UCB-based selection and regret-based scheduling. Specifically, we set up the two variants of BELIEFFUZZ, BELIEFFUZZ-*r* and BELIEFFUZZ-*c*, which disable the UCB-based selection and the regret-based scheduling, respectively, and rerun the previous experiments.

Figure 9 shows the experimental results. We can observe that BELIEFFUZZ achieves 12.18% and 15.02% more coverage than BELIEFFUZZ-*r* and BELIEFFUZZ-*c* on average, with 41.24% and 43.25% time reduction for achieving the same coverage in 24 hours experiment. This result demonstrates the significance and the necessity of the two strategies, as both of them indeed contribute to the efficiency of seed scheduling, and their combination allows us to achieve higher coverage with less time consumption.

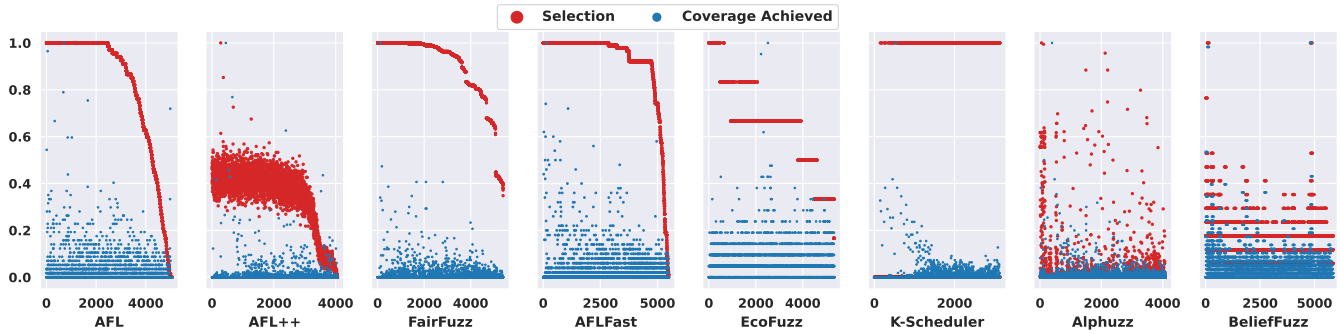


Figure 10: Distributions of the coverage achieved by each seed and its frequency of being selected in each fuzzer. We demonstrate the average benefit/cost proportion when fuzzing program *objdump* in 24 hours experiment. The x-axis is the index of seeds in the discovering order. The y-axes are the proportions of the coverage achieved and the number of selections of each seed, illustrated in blue dots and red crosses, respectively.

Table 3: Bugs detected by different fuzzers.

	AFL	AFL++	FairFuzz	AFLFast	EcoFuzz	K-Scheduler	Alphuzz	BeliefFuzz
mp42aac	1	1	2	2	1	2	1	3
mp4info	8	9	0	8	8	5	8	9
nm-new	0	0	0	0	0	0	0	0
objdump	0	0	0	0	0	0	0	1
faust	2	1	1	2	3	3	0	3
fig2dev	0	0	0	0	0	0	0	2
mp4box	0	0	0	0	0	0	0	20
magick	0	0	0	0	0	0	0	0
djpeg	0	0	0	0	0	0	0	0
tiff2pdf	0	0	0	0	0	0	0	0
tiff2ps	0	0	0	0	0	0	0	2
tiffcrop	0	1	0	2	0	1	0	7
Total	11	12	3	14	12	11	9	47

Table 4: Details of Bugs detected by BeliefFuzz.

Program	Reported Bugs
mp42aac	issue #637-639
mp4info	issue #622-623, 630, 640-642, 645, 651-652
objdump	bug #28247
faust	issue #655-657
fig2dev	ticket #125-126
mp4box	issue #1884, 1886-1893, 1895-1905
tiff2ps	issue #267-268
tiffcrop	issue #269-273, 275, 291

4.3.1 *Does the Selection of Each Seed Meet the Expectation to Find Coverage?*

To show how the adaptive UCB used in BeliefFuzz helps select the seed that can achieve higher coverage, we evaluated the benefit/cost ratio of each seed measured by the number of new paths found and the total number of selections. We used the same evaluation configurations used previously with ten repeat experiments and studied the distributions of the two aspects mentioned.

Table 5 shows the experiment results. Ideally, the fuzzer should prioritize the seeds that can achieve higher coverage. Therefore, the distribution of coverage achieved by each seed should be close to the number of selections. We used the Jensen–Shannon divergence (JS divergence) [26] to measure this similarity between the two distributions. Overall, BeliefFuzz has the smallest JS divergence, $3.79E$-02, compared with the state-of-the-art fuzzer on average and outperforms in all the evaluated projects. This statistic represents the effectiveness of UCB-based scheduling, which makes the seed chosen to find expected coverage without resource waste.

We further studied one case to illustrate such effectiveness using the results of *objdump* shown in Figure 10. The

Table 5: The Jensen–Shannon divergence between the two distributions of the number of selections spent and the coverage achieved for each seed. The smaller the divergence is, the more similar the two distributions are. *n/a* represents the occurrence of the functionality issues.

	AFL	AFL++	FairFuzz	AFLFast	EcoFuzz	K-Scheduler	Alphuzz	BeliefFuzz
mp42aac	0.269	0.215	0.327	0.273	0.107	0.271	0.178	0.058
mp4info	0.283	0.278	0.316	0.276	0.120	0.297	0.226	0.037
nm-new	0.339	0.294	0.298	0.278	0.161	0.384	0.249	0.021
objdump	0.413	0.317	0.362	0.345	0.143	0.432	0.233	0.015
faust	0.515	0.481	0.568	0.395	0.139	0.413	0.183	0.012
fig2dev	0.211	0.579	0.464	0.237	0.126	0.502	0.278	0.079
mp4box	0.421	0.442	0.419	0.407	0.092	0.491	0.281	0.026
magick	0.444	0.405	0.456	0.452	0.131	0.516	0.327	0.012
djpeg	0.433	0.370	0.514	0.497	0.182	0.422	0.303	0.061
tiff2pdf	0.362	0.474	0.626	0.364	0.177	0.544	0.307	0.070
tiff2ps	0.368	0.420	0.536	0.363	0.165	0.448	0.284	0.045
tiffcrop	0.389	0.302	0.461	0.370	0.086	0.417	0,274	0.019
Avg.	**0.371**	**0.381**	**0.446**	**0.355**	**0.135**	**0.428**	**0.260**	**0.038**

distributions demonstrate that BeliefFuzz does improve the effectiveness of the seed scheduling in making each seed achieve the expected coverage benefit since the two distributions are close to each other. On the contrary, all benchmark fuzzers stick to partial seeds with an extremely high selection frequency and thus eventually cannot efficiently achieve high coverage. We find that BeliefFuzz selects the seed more evenly to explore the program thoroughly. Thus, BeliefFuzz can find more coverage overall.

Surprisingly, we notice that the four baselines, AFL, AFLFast, FairFuzz, and K-Scheduler find new coverage using a minority of the seeds ($< 20\%$) in 24-hour experiments while all seeds are selected more evenly than other fuzzers. For K-Scheduler, it almost selects the seeds evenly, which makes most of the mutations unable to detect new coverage, and some seeds have never been selected during the 24-hour experiment. Its largest JS divergence also indicates that such a selection mechanism cannot simultaneously balance the cost and benefit. We observe one of the possible reasons is that an enormous size of the programs may introduce bias for calculating the potential for each seed using graph centrality. This result proves the importance of the seed scheduling mechanism. It also demonstrates the significance of the next evaluated component, regret-based power scheduling, to terminate the mutation that cannot find any news coverage at an early stage.

Table 6: The proportions of the mutation iterations not achieving new coverage in the evaluated fuzzers. The results come from 24-hour experiments averaged over ten runs. Noticed that all these iterations in BELIEFFUZZ are terminated earlier with the regret-based power scheduling.

	AFL	AFL++	FairFuzz	AFLFast	EcoFuzz	K-Scheduler	Alphuzz	BELIEFFUZZ
mp42aac	99.48%	98.93%	99.91%	99.85%	84.64%	71.69%	87.21%	37.22%
mp4info	99.42%	99.40%	99.92%	99.85%	84.58%	66.75%	82.13%	57.38%
nm-new	99.34%	98.76%	99.93%	99.67%	79.73%	70.36%	82.29%	64.62%
objdump	99.40%	98.65%	99.95%	99.68%	83.51%	81.23%	79.34%	74.98%
faust	98.93%	98.73%	99.81%	98.86%	75.60%	72.42%	81.83%	71.16%
fig2dev	99.71%	98.74%	99.90%	99.82%	86.02%	96.42%	89.77%	53.68%
mp4box	99.24%	99.57%	99.24%	98.22%	78.81%	93.15%	74.90%	69.02%
magick	99.12%	98.23%	99.88%	99.58%	81.37%	72.46%	81.12%	68.13%
djpeg	99.38%	98.60%	99.96%	99.77%	88.16%	76.11%	83.71%	57.09%
tiff2pdf	99.53%	99.00%	99.97%	99.85%	84.15%	88.68%	87.32%	70.02%
tiff2ps	99.56%	99.36%	99.97%	99.84%	86.06%	86.27%	86.49%	63.45%
tiffcrop	99.54%	98.21%	99.95%	99.81%	99.96%	93.23%	82.09%	74.47%
Avg.	**99.39%**	**98.85%**	**99.87%**	**99.65%**	**84.38%**	**80.73%**	**83.18%**	**63.43%**

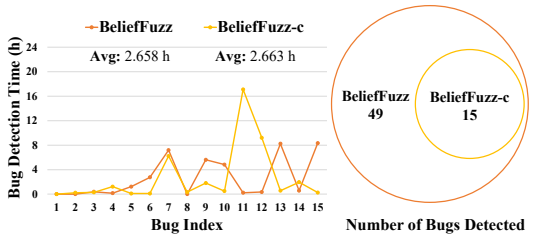


Figure 11: The impact of regret-based scheduling for bug detection. The x-axis is the index of bugs found in the previous experiments in both BELIEFFUZZ and BELIEFFUZZ-c. The y-axis is the time used for detection.

4.3.2 Can Regret-based Scheduling Reduce Energy Waste?

Regret-based scheduling stops the fuzzing iteration if no coverage is found using the expected number of mutations. To understand the effectiveness of the termination, we first evaluated the number of mutation iterations that do not contribute any coverage in the benchmark fuzzers. Then, we examined the proportion of the mutation iterations terminated by the regret-based energy assignment in BELIEFFUZZ. We used the same evaluation configuration as Section 4.2 and ran each fuzzer in the evaluation benchmark mentioned in Table 2 with a time budget of 24 hours.

The results are shown in Table 6. Overall, a majority of the mutation iterations (>90% on average) in existing fuzzers, cannot contribute new coverage to the results, which denotes a huge resource waste in fuzzing. This statistic also explains why all existing work cannot explore the majority of the discovered seeds even once in previous experiments shown in Figure 10. They all excessively allocate the energy to the seeds selected without any insurance if the selection is not ideal. Fortunately, BELIEFFUZZ can stop these useless mutation iterations earlier to improve efficiency. Overall, BELIEFFUZZ has the lowest failure rate (63.43%) on average of finding new coverage for each selected seed, further indicating the effectiveness of seed selection. Therefore, BELIEFFUZZ can efficiently switch to promising seeds for achieving higher coverage.

Moreover, we further show the improvement of regret-based scheduling by evaluating the different time costs for bug detection. The results shown in Figure 11 indicate regret-based scheduling can help BELIEFFUZZ detect 34 more bugs, Even though BELIEFFUZZ has almost equivalent performance compared with BELIEFFUZZ-c for those detected bugs, the overall outcomes demonstrate that the early termination of mutating unproductive seeds helps the fuzzer detect more bugs.

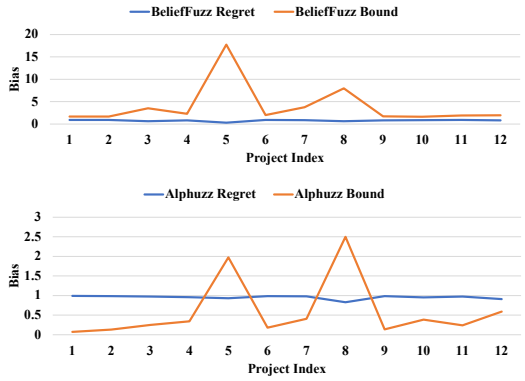


Figure 12: Empirical evaluation of the guarantee in BELIEFFUZZ-c and Alphuzz. The x-axis represents the program identified using the index shown in Table 2. The y-axis is the value calculated for the regret and bound based on Theorem 1.

4.4 Empirical Evidence of the Guarantee

The usage of the MCUCB model can provide a bound for the application scenario, which can indicate that the potential coverage loss caused by non-ideal selection is bounded. However, it is impractical to verify the existence of the bound in Theorem 1 since we cannot obtain the ideal benefit, $benefit^*$. Therefore, we want to study whether the bound exists empirically. Specifically, we conduct the experiment to verify the existence of the guarantee as shown below:

$$|\mathbb{E}(\overline{benefit}_n) - benefit^*| \leq \mathcal{O}(\frac{\ln n}{n})$$

where $benefit^*$ is the $benefit$ averaged over all ideal selections.

To measure the existence of the bound, we leverage several approximations for the ideal scenarios. First, we assume that a fuzzer can select the seeds using perfect information, which means that all selected seeds can find a new path. Next, we collect the relevant statistics, the actual coverage achieved for $benefit^*$, and the selection times, n, in fuzzing for calculating Theorem 1. Specifically, we use

$$|\mathbb{E}(\overline{benefit}_n) - benefit^*| = |1 - \frac{Coverage_{Achieved}}{n}|$$

$Coverage_{Achieved}$ is the overall coverage achieved while mutating the n selected seeds. We regard $\mathcal{O}(\frac{\ln n}{n})$ as $\frac{\ln n}{n}$ to simplify the formula computation.

We evaluate the final results and real-time performance in 24-hour experiments ten times, respectively. The average results shown in Figure 12 indicate the existence of the guarantee since all regrets are less than the maximal bound calculated by the theorem in all evaluated projects. As the bound represents the maximal bias toward the ideal selection, the difference between the regret and the bound represents the ability of the selection mechanism. The larger the distance

is, the better the selection approach is. Meanwhile, we find out that Alphuzz indeed does not have a guarantee because of the conventional Monte Carlo method.

Overall, the existence of the guarantee indicates BELIEFFUZZ can balance the cost and the benefit simultaneously to improve fuzzing, which the maximal bias of the coverage achieved compared with the one using the ideal selection is now computable. Meanwhile, the guarantee provides an expectation of the potential coverage that can be achieved so that we can use it to arrange the time budget spent for fuzzing. Specifically, we can predict the least coverage achieved, $benefit^*$, by fuzzer using the number of selections, n, when assuming the ideal selection can achieve the highest coverage, which the average benefit, $\mathbb{E}(\overline{benefit}_n)$, is 1.

Moreover, the guarantee can help quantify the improvement of future work. As the guarantee provides a computable approximation for fuzzing, future work can use it to analyze their optimization theoretically based on the improvement of the guarantee. Specifically, fuzzing can have a better selection mechanism to enhance the performance along with the minimization of the maximal bound.

4.5 Discussion

Benefit Beyond Coverage. In BELIEFFUZZ, the UCB consists of the benefit based on the potential coverage achievable for each seed. This metric is designed for current coverage-guided fuzzing. Nonetheless, according to the concept in MCP, the benefit is adapted from the problem domains, which provide the possibility to extend BELIEFFUZZ for other application scenarios. For example, we can design the benefit with high-level semantics, e.g., event dependence in system calls in the operating system. Other semantics, such as program state machine or automata and function similarity, can also help fuzzing detect different kinds of program behaviors. Meanwhile, it also supports integrating fuzzing with other techniques such as concolic execution. Since various analysis methods are available for examining the target programs, their combination can help design a more robust benefit as guidance for further analysis.

Combining with Input Generation. Conventionally, input generation and seed scheduling are the two main factors dominating the fuzzing performance. However, the majority of the existing work improves them independently, which could not make their optimized method generalized and extendable for future work. The model of MCUCB used in BELIEFFUZZ currently adapts its basic form mentioned in Section 2, which regards the input generation as the *default policy* of the tree. Some existing theories are attempting to improve the dynamic simulation by the scheduling strategies [9], [13], [14]. It could be the future direction for fuzzing to simultaneously improve input generation and seed scheduling for higher performance.

Threats to Validity. The main concern is the influence of randomness in the input generation. Even though we have conducted the experiments multiple times for fairness, different input sequences might produce fluctuated outcomes in other projects. Still, the results meet the expectation that BELIEFFUZZ achieves higher coverage than the benchmark fuzzers with less time cost and the number of executions.

5 RELATED WORK

Application-based Seed Scheduling. The goal of fuzzing is to find more vulnerabilities in a given program. Therefore, coverage may not be the only metric to measure progress and various fitness functions have been designed. Directed fuzzing, which aims to reproduce a targeted vulnerability residing at specific parts of a program, utilizes different distance metrics [6], [10] to measure the closeness of the executed seed toward the vulnerable program point. Meanwhile, directed fuzzing also terminates the execution earlier if it cannot meet the expectation to achieve the goal. FuzzGuard [3] leverages the deep learning model to predict the reachability before the execution. Beacon [18] stops the execution whenever it violates the precondition inferred to trigger the target vulnerabilities. Performance-oriented fuzzing [23], [30] prefers the seed that triggers more execution time cost. Similarly, the amount of memory usage is set as the fitness function for finding exhaustive memory usage issues [38]. Furthermore, instead of these concrete metrics, abstract state machines that model the high-level semantics can also prioritize the seeds. For example, UAFL [36] generates a state machine for the use-after-free vulnerability. Then, the seeds that explore more states are prioritized for further exploration. Overall, designing a precise benefit function for fuzzing is still valuable, but BELIEFFUZZ allows extending the definition of benefit and still can provide the guarantee no matter what benefit is provided to avoid falling into the local optima.

Scheduling for Parallel Fuzzing. The other problem closely related to seed scheduling is prioritizing seeds in multiple cooperated fuzzing techniques. As mentioned earlier, the two most common assistant techniques, taint analysis and concolic execution, are notorious for their scalability issues. Therefore, existing efforts attempt to find the most valuable seed for making full use of these precise but expensive methods. To measure the value of the seeds, Digfuzz [40] regards the execution frequency of each path prefix as its difficulty and chooses the most difficult one for symbolic execution. ParmeSan [29] considers the number of reachable checks reported by a sanitizer [32] to be the potential of seed to find more bugs. Furthermore, Enfuzz [12] combines multiple state-of-the-art fuzzers into one platform and makes their seed queues cooperate in taking the benefits from all existing work simultaneously. On the contrary, BELIEFFUZZ focuses on improving an individual fuzzer, and we will extend the scheduling scheme to support multiple fuzzers in future work.

6 CONCLUSION

We have presented BELIEFFUZZ, which transforms the seed scheduling procedure in fuzzing into a Monte-Carlo planning process. It prevents the seed scheduling procedure from falling into the high benefit/cost traps. Compared with existing greybox fuzzers, BELIEFFUZZ is more efficient than the state-of-the-art greybox fuzzers with 3.22x speedup, and such efficiency improvement also enables effectiveness enhancement with a 1.56x coverage increase.

REFERENCES

[1] "Afl: american fuzzy lop," http://lcamtuf.coredump.cx/afl/, 2013, accessed: 2013.

[2] "MOPT: Optimized mutation scheduling for fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, 2019. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/lyu

[3] "Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning," in *29th USENIX Security Symposium (USENIX Security 20)*. Boston, MA: USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/zong

[4] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: fuzzing with input-to-state correspondence," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/

[5] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2, pp. 235–256, 2002.

[6] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 2329–2344. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134020

[7] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 1032–1043. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978428

[8] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[9] G. Chaslot, C. Fiter, J.-B. Hoock, A. Rimmel, and O. Teytaud, "Adding expert knowledge and exploration in monte-carlo tree search," in *Advances in Computer Games*, H. J. van den Herik and P. Spronck, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–13.

[10] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 2095–2108. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243849

[11] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, "SAVIOR: towards bug-driven hybrid testing," *CoRR*, vol. abs/1906.07327, 2019. [Online]. Available: http://arxiv.org/abs/1906.07327

[12] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19. USA: USENIX Association, 2019, p. 1967–1983.

[13] H. Finnsson and Y. Björnsson, "Simulation-based approach to general game playing." in *Aaai*, vol. 8, 2008, pp. 259–264.

[14] ——, "Learning simulation control in general game-playing agents," in *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[15] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[16] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, 2018, pp. 679–696. [Online]. Available: https://doi.org/10.1109/SP.2018.00040

[17] C. J. Geyer, "Practical markov chain monte carlo," *Statistical science*, pp. 473–483, 1992.

[18] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "Beacon : Directed grey-box fuzzing with provable path pruning," in *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2022, pp. 104–118. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46214.2022.00007

[19] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Machine Learning: ECML 2006*, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 282–293.

[20] L. Kocsis, C. Szepesvári, and J. Willemson, "Improved monte-carlo search," *Univ. Tartu, Estonia, Tech. Rep*, vol. 1, 2006.

[21] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis and transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO '04. USA: IEEE Computer Society, 2004, p. 75.

[22] P. R. G. Layard *et al.*, *Cost-benefit analysis*. Cambridge University Press, 1994.

[23] C. Lemieux, R. Padhye, K. Sen, and D. Song, "Perffuzz: Automatically generating pathological inputs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: ACM, 2018, pp. 254–265. [Online]. Available: http://doi.acm.org/10.1145/3213846.3213874

[24] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 475–485. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238176

[25] Y. Li, Y. Xue, H. Chen, X. Wu, C. Zhang, X. Xie, H. Wang, and Y. Liu, "Cerebro: Context-aware adaptive fuzzing for effective vulnerability detection," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 533–544. [Online]. Available: https://doi.org/10.1145/3338906.3338975

[26] J. Lin, "Divergence measures based on the shannon entropy," *IEEE Transactions on Information theory*, vol. 37, no. 1, pp. 145–151, 1991.

[27] P. E. McKnight and J. Najab, *Mann-Whitney U Test*. American Cancer Society, 2010, pp. 1–1. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470479216.corpsy0524

[28] N. Metropolis and S. Ulam, "The monte carlo method," *Journal of the American Statistical Association*, vol. 44, no. 247, pp. 335–341, 1949, pMID: 18139350. [Online]. Available: https://www.tandfonline.com/doi/abs/10.1080/01621459.1949.10483310

[29] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "Parmesan: Sanitizer-guided greybox fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2289–2306. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/osterlund

[30] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 2155–2168. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134073

[31] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 861–875. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/rebert

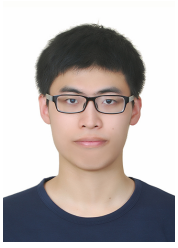
[32] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *USENIX ATC 2012*, 2012. [Online]. Available: https://www.usenix.org/conference/usenixfederatedconferencesweek/addresssanitizer-fast-address-sanity-checker

[33] D. She, A. Shah, and S. Jana, "Effective seed scheduling for fuzzing with graph centrality analysis," in *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2022, pp. 1558–1558. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46214.2022.00129

[34] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

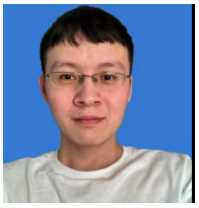
[35] D. Silver and J. Veness, "Monte-carlo planning in large pomdps," *Advances in neural information processing systems*, vol. 23, 2010.
[36] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen, and Y. Sui, "Typestate-guided fuzzer for discovering use-after-free vulnerabilities," in *Proceedings of the 42nd International Conference on Software Engineering*, 2020.
[37] J. Wang, C. Song, and H. Yin, "Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing," in *29th Annual Network and Distributed System Security Symposium, NDSS 2022*, 2022.
[38] C. Wen, H. Wang, Y. Li, S. Qin, L. Yang, X. Zhiwu, C. Hongxu, X. Xiaofei, P. Geguang, and L. Ting, "Memlock: Memory usage guided fuzzing," in *Proceedings of the 42nd International Conference on Software Engineering*, 2020.
[39] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou, "Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2307–2324. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/yue
[40] L. Zhao, Y. Duan, H. Yin, and J. Xuan, "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*, 2019.
[41] Y. Zhao, X. Wang, L. Zhao, Y. Cheng, and H. Yin, "Alphuzz: Monte carlo search on seed-mutation tree for coverage-guided fuzzing," in *Proceedings of the 38th Annual Computer Security Applications Conference*, ser. ACSAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 534–547. [Online]. Available: https://doi.org/10.1145/3564625.3564660

Heqing Huang Heqing Huang is a Postdoc Research Fellow in the Department of Computer Science and Engineering, the Hong Kong University of Science and Technology. His major research interest is to enhancing fuzz testing with various program analysis techniques to ensure software security and reliability. He has published multiple papers at the premium venues of security, software engineering, and programming languages. His research has achieved a significant industrial impact, which has been successfully deployed in Huawei and awarded by distinguished collaborator award. His research also successfully detected hundreds of software vulnerabilities in open-source software and received many acknowledgments from various communities, such as GNU, Therome provers, and Linux Kernel. Heqing Huang obtained his Ph.D., M.Sc, and B.S. from the Hong Kong University of Science and Technology, Imperial College London, and Xidian University, respectively.



Hung-Chun Chiu HungChun Chiu is a Ph.D. candidate in the Department of Computer Science Engineering, Hong Kong University of Science and Technology. His current research interest is improving fuzzing performance by static analysis and symbolic execution. He received his BEng and Mphil degrees from the School of Software, Tsinghua University, China.

Qingkai Shi Qingkai Shi is a Postdoc Research Associate with the department of computer science at Purdue University. His major research interest is the use of compiler techniques to ensure software reliability. He has published extensively at premium venues of programming languages, software engineering, and cybersecurity. His research received many awards including ACM SIGSOFT Distinguished Paper Award, ACM SIGPLAN Distinguished Paper Award, and Hong Kong Ph.D. Fellowship. His research has led to the discovery of over a hundred software vulnerabilities in open-source software and has been successfully commercialized in Sourcebrella Inc, a static analysis tool vendor. Qingkai obtained his B.S. and Ph.D. from Nanjing University and the Hong Kong University of Science and Technology, respectively. This work was done when he was with the Hong Kong University of Science and Technology.



Peisen Yao Peisen Yao is an Assistant Professon in the Colledge of Computer Science and Technology, Zhejiang University. He received his Ph.D, degree from The Hong Kong University of Science and Technoogy. His major research interests include program analysis, program syntheis, and automated reasoning. More information is available on https://rainoftime.github.io/

Charles Zhang Charles Zhang is an Associate Professor and the director of the Cybersecurity Lab in the Department of Computer Science and Engineering, the Hong Kong University of Science and Technology (HKUST). His major research interest is using program analysis techniques to improve software reliability. He publishes extensively in premium conferences and journals of programming languages, software engineering, and security. He has served as an associate editor of IEEE TSE and on many organizational and technical committees of leading international conferences. His research awards include the distinguished paper awards from leading conferences such as PLDI, OOPSLA, and ICSE, as well as the ACM SIGSOFT Doctoral Dissertation Award. He has also generated a significant industrial impact including the successful commercialization of the Pinpoint static analysis tool through Sourcebrella Inc, acquired by the Ant Group, and winning twice the Huawei distinguished collaborator award. He has also worked as a software engineer at Motorola Inc, served as an expert advisor to Huawei Inc and as an expert security panelist of the Hong Kong Monetary Authority. His research is supported by Research Grant Council, Innovation and Technology Fund, and grants from Huawei, TCL, Microsoft, and IBM. Charles obtained his Ph.D, M.Sc, and B.Sc. with honours, all from University of Toronto.