Learning Seed-Adaptive Mutation Strategies for Greybox Fuzzing

Myungho Lee
Korea University
myungho_lee@korea.ac.kr

Sooyoung Cha*
Sungkyunkwan University
sooyoung.cha@skku.edu

Hakjoo Oh*
Korea University
hakjoo_oh@korea.ac.kr

***Abstract*—In this paper, we present a technique for learning seed-adaptive mutation strategies for fuzzers. The performance of mutation-based fuzzers highly depends on the mutation strategy that specifies the probability distribution of selecting mutation methods. As a result, developing an effective mutation strategy has received much attention recently, and program-adaptive techniques, which observe the behavior of the target program to learn the optimized mutation strategy per program, have become a trending approach to achieve better performance. They, however, still have a major limitation; they disregard the impacts of different characteristics of seed inputs which can lead to explore deeper program locations. To address this limitation, we present SEAMFUZZ, a novel fuzzing technique that automatically captures the characteristics of individual seed inputs and applies different mutation strategies for different seed inputs. By capturing the syntactic and semantic similarities between seed inputs, SEAMFUZZ clusters them into proper groups and learns effective mutation strategies tailored for each seed cluster by using the customized Thompson sampling algorithm. Experimental results show that SEAMFUZZ improves both the path-discovering and bug-finding abilities of state-of-the-art fuzzers on real-world programs.**

I. INTRODUCTION

Mutation-based greybox fuzzing is one of the most popular techniques for finding software vulnerabilities [4], [31], [34], [3], [14]. Without any prior knowledge on the target program, greybox fuzzing can generate a huge number of test-cases by repeating the following three steps: seed selection, seed mutation, and execution. Taking as input a target program and a seed corpus, it selects a seed input to be mutated, generates test-cases by mutating the selected seed input in various ways, and executes the target program with those generated mutants. By repeating these steps, mutation-based fuzzers can automatically test the target program with numerous different test-cases in order to explore deeper program locations and detect vulnerabilities.

The performance of greybox fuzzers heavily depends on the mutation strategy. In general, mutation-based fuzzing tools [49], [41], [34], [12] maintain a set of predefined mutation methods that specify where to mutate and how to mutate. These tools also have their own decision-making strategies on selecting the *mutation methods* for test-case generation, so-called mutation strategy. As test-cases are generated by mutating the seed input, the mutation strategy is critical for the performance of fuzzers and therefore many techniques have been proposed to improve its effectiveness. AFL [49], for example, includes various types of mutation operators (e.g., 1-bit flipping, addition with random value, and bytes replacement) and selects them randomly to generate test-cases. VUZZER [34] has two mutation methods, namely crossover and mutation, and chooses them with a fixed probability.

Existing Approaches. Recently, program-adaptive mutation strategies [28], [20], [46], [23], [6], [47] have been spotlighted as state-of-the-art techniques for mutation-based fuzzers. Even with significant effort on improving the mutation strategy, existing program-agnostic fuzzers such as AFL [49], HONGFUZZ [41], and VUZZER [34] only follow a fixed and pre-defined probability distribution to select mutation methods regardless of the target program, which potentially degrades the overall fuzzing performance. The program-adaptive approach aims to overcome this limitation by learning a probability distribution of selecting mutation methods optimized for each target program. Notably, MOPT [28] is the current state-of-the-art technique that can adaptively find an optimal mutation strategy during the fuzzing process by employing a learning algorithm based on particle swarm optimization.

Those program-adaptive techniques, however, have a major shortcoming; they aim to learn the effective mutation strategies regarding to only the target program, yet disregard the impacts of different characteristics of each seed input. Since the program-adaptive approaches primarily observe the behaviors triggered by the characteristics shared over the majority of seed inputs, they often miss the individual characteristics of each seed input which differ from the majority. These differences, however, often play a key role in satisfying specific conditions of the program to reach the deep-seated program locations. Hence, capturing and maintaining the differences of each seed input are inevitably important in fuzzing to penetrate deeper program locations and find subtle bugs.

SEAMFUZZ. In this paper, we present SEAMFUZZ, a grey-box fuzzer with a novel seed-adaptive mutation strategy. Unlike the program-adaptive techniques such as MOPT [28], SEAMFUZZ aims to capture individual characteristics of seed inputs and apply different mutation strategies for different seed inputs. The main technical challenge to achieve this goal is how to effectively learn and maintain multiple probability distributions that are associated with different seed inputs. In this paper, we address this challenge with two algorithms: a clustering algorithm and an online learning algorithm. SEAMFUZZ clusters seed inputs into groups based on their

*Corresponding authors.

syntactic and semantic properties and learns online different probability distribution of selecting mutation methods per seed cluster by leveraging a variant of Thompson sampling [1]. The learned probability distribution associated with each cluster is used to sample effective mutation methods when mutating the seed inputs that belong to the cluster.

The experimental results show that SEAMFUZZ significantly improves the performance of existing state-of-the-art mutation-based fuzzers, AFL++ [12] and MOPT [28]. We have implemented SEAMFUZZ and evaluated its effectiveness on 14 real-world programs with a total size of 6.8 millions of lines of code in comparison with AFL++ and MOPT. On bug-detection ability, SEAMFUZZ successfully found 27 unique bugs with crash types and states which AFL++ and MOPT totally failed to detect, and generated 56.4% and 57.1% more crash inputs compared to AFL++ and MOPT on average, respectively. In terms of branch coverage, SEAMFUZZ covered 5.6% and 7.7% more branches than AFL++ and MOPT, respectively, on average. The experimental results also show that our clustering algorithm for assigning each seed to proper groups and our variant of Thompson sampling for learning effective mutation strategy are essential for enhancing the overall performance of fuzzer.

Contributions. We summarize our contributions below:

- We present a new seed-adaptive approach for mutation-based fuzzing. To our knowledge, SEAMFUZZ is the first work that raises the need for the transition from the program-adaptive approaches [28], [20], [46], [23], [6] to seed-adaptive approaches for selecting effective mutation methods.
- We present an effective algorithm for learning seed-adaptive mutation strategies. The key idea is to cluster the seed inputs into proper groups based on syntactic/semantic characteristics and learn different probability distributions of selecting mutation methods for each seed cluster by customizing the Thompson sampling algorithm.
- We demonstrate the effectiveness of SEAMFUZZ by evaluating it on 14 programs and comparing its performance with the current state-of-the-art program-adaptive fuzzer. We make our tool, SEAMFUZZ, open-sourced and all the benchmarks publicly available.[1]

II. PRELIMINARIES

We provide background necessary to develop our technique.

A. Mutation-based Fuzzing

In general, mutation-based greybox fuzzers are comprised of the three components as shown in Figure 1: selection, mutation, and execution. The first two components, seed selection and seed mutation, especially have major impacts on the performance of fuzzing. Hence, each mutation-based greybox fuzzer has its own strategies for them, namely seed selection strategy and seed mutation strategy, to maximize its performance.

[1]SEAMFUZZ: https://github.com/kupl/SeamFuzz-public

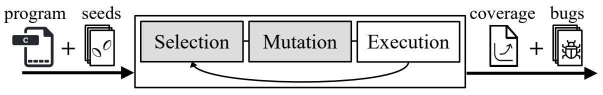

Fig. 1. Workflow of Mutation-based Greybox Fuzzer

Algorithm 1 Mutation-based Greybox Fuzzing

```
Input: Program $pgm$, seed input corpus $I$.
Output: path coverage $Cover(I)$, crashing input corpus $Crash$
 1: procedure AFL($pgm, I$)
 2:     $Crash \leftarrow \emptyset$
 3:     repeat
 4:         $s \leftarrow$ SELECTSEED($I$)
 5:         for $i = 1$ to $n$ do
 6:             $M \leftarrow$ SAMPLE($\mathbf{P}_r$)     $\triangleright\ M = \langle(\mathsf{op}_0, \mathsf{loc}_0), ..., (\mathsf{op}_k, \mathsf{loc}_k)\rangle$
 7:             $s' \leftarrow$ MUTATE($s, M$)
 8:             $(I, Crash) \leftarrow$ EXECUTE($pgm, s', I, Crash$)
 9:     until $timeout$ reached
10:     return $(Cover(I), Crash)$
```

A seed selection strategy determines how to select a seed input to mutate [4], [26], [35], [14]. Generally, a mutation-based fuzzer maintains a seed corpus which can be accumulated with newly generated test-cases at runtime, and selects a seed input to mutate from the corpus. These processes are based on specific criteria defined by each mutation-based fuzzing tool, namely seed selection strategy.

A seed mutation strategy decides how to mutate the selected seed input for test-case generation [28], [20], [46], [23], [6]. In general, mutation-based greybox fuzzer has a set of predefined mutation methods; each mutation method consists of the operator (op) and the location (loc) that specify how to mutate and where to mutate, respectively. During the mutation step, a mutation-based greybox fuzzer selects mutation methods by following a certain probability distribution of choosing the mutation methods. AFL, for example, maintains 11 types of mutation operators (e.g., flipping bit, arithmetic increments, block manipulation) and selects them by following the uniform probability distribution; for the mutation location, it basically selects random bytes of the given seed input.

B. Seed Mutation

Let us delve into more details on the mutation strategy of AFL, which both MOPT [28] and SEAMFUZZ build upon. AFL first selects a seed input s to mutate from the seed input corpus I at line 4 in Algorithm 1. Then, at line 5 in Algorithm 1, AFL randomly decides the number n of test-cases to generate before starting test-case generation. The typical value for n in practice ranges between 16 and 65,536 (see AFL [49] for details). As the first step in the loop for test-case generation, it formulates a vector (M) of mutation methods in which an element ($m = (\mathsf{op}, \mathsf{loc})$) specifies how to mutate (op) and where to mutate (loc). AFL selects each mutation method (m) of M by following the uniform probability distribution (denoted $\mathbf{P}_r$).

After formulating M, AFL mutates the seed input s by applying each mutation method in M sequentially to seed

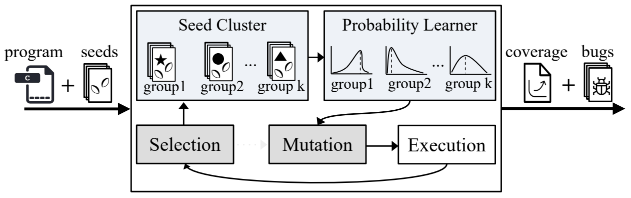

Fig. 2. Workflow of SeamFuzz

input s. For example, let us assume that the seed input s has the size of 2 bytes (16 bits) and M is as follows:

$$\begin{array}{rcl} s &=& \text{"0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1"}, \\ M &=& \langle (flip_{1bit}, 1), (flip_{2bits}, 10) \rangle \end{array} \quad (1)$$

where the seed input s is expressed as a string of bits and each element of M represents the selected mutation method m (i.e., a pair of op and loc). With this M, AFL flips the first bit of s (i.e., 0 to 1) and the consecutive two bits from the 10-th position (i.e., 10 to 01) to generate a new test-case s' ("**1** 0 0 0 0 0 0 0 0 **0 1** 0 0 0 0 1").

Using the newly generated test-case s' as an input, AFL executes the target program pgm as shown at line 8 in Algorithm 1. s' is accumulated to the seed input corpus I if it covers a new execution path or makes a program crash. In this paper, we refer to these test-cases as interesting test-cases. AFL repeats these steps until it generates and tests n test-cases. Undoubtedly, the methodology for formulating the mutation methods M, the mutation strategy, greatly affects the performance of fuzzing tools.

Limitation. Traditional program-agnostic fuzzers such as AFL totally rely on the randomness of selecting mutation methods for test-case generation, which is a major shortcoming preventing optimal performance. It is well-known that AFL can generate enormous number of test-cases within a reasonable time (e.g., 96 millions of test-cases generated in 3 days) for any types of testing program. It, however, shows low efficiency of generating meaningful test-cases, which lead to unexplored program locations, due to its inherent randomness.

In order to improve the efficiency and the performance of mutation-based fuzzers, the program-adaptive approaches, notably MOpt [28], have become a trending approach, which aims to control the probability distribution for selecting mutation methods with respect to the target program. However, they tune the mutation strategy with respect to only the testing program, missing the opportunity of leveraging the different characteristics of distinct seed inputs. We experimentally observed that some seed inputs can be effectively mutated for penetrating into deeper program locations or triggering program crashes if it follows a certain probability distribution far from the one learned on a target program. This observation motivated us to develop a new technique which automatically applies the proper probability distribution of choosing mutation methods to different seed inputs.

Algorithm 2 SeamFuzz

```
Input: Program $pgm$, seed input corpus $I$.
Output: path coverage $Cover(I)$, crashing input corpus $Crash$
1:  procedure SeamFuzz($pgm, I$)
2:      $Crash \leftarrow \emptyset$
3:      $G \leftarrow \emptyset$                          ▷ information for seed groups
4:      repeat
5:          $s \leftarrow$ SelectSeed($I$)
6:
7:          $g \leftarrow$ Cluster($s, G$)            ▷ $g = (s_{\mathsf{id}}, S, \mathbf{P}_{\mathsf{id}}, D_{\mathsf{g}}, D_{\mathsf{b}})$
8:          $G \leftarrow G \backslash \{g\}$
9:
10:         for $i = 1$ to $n$ do
11:             $\mathbf{P} \leftarrow$ sample from $\{\mathbf{P}_{\mathsf{id}}, \mathbf{P}_r\}$ with prob = $[\epsilon, 1-\epsilon]$
12:             $M \leftarrow$ Sample($\mathbf{P}$)      ▷ $M = \langle (\mathsf{op}_0, \mathsf{loc}_0), ..., (\mathsf{op}_k, \mathsf{loc}_k) \rangle$
13:             $s' \leftarrow$ Mutate($s, M$)
14:             $(I, Crash, g) \leftarrow$ Execute($pgm, s', g$)
15:         $g' \leftarrow$ Learn($g$)
16:         $G \leftarrow G \cup \{g'\}$
17:     until $timeout$ reached
18:     return $(Cover(I), Crash)$
```

III. Our Technique

In this section, we present our technique for automatically learning optimized probability distributions of choosing effective mutation methods regarding to each seed input. A naive approach to achieve this goal would be to learn probability distributions for all seed inputs individually. In practice, however, this simple approach is not appropriate to test real-world programs as we demonstrate in Section IV-D. This is because the number of seed inputs maintained during fuzzing is typically tens of thousands, so there is little chance that the individual probabilities learned for each input are reused. To address this challenge, our algorithm comprises of two key components: seed cluster and probability learner.

A. *Overview of* SeamFuzz

Let us first briefly explain how our approach works. Figure 2 illustrates the workflow of SeamFuzz. SeamFuzz comprises two more phases compared to the traditional mutation-based fuzzing tool: (1) **seed cluster** which clusters the seed input into the seed group with the similar characteristics (Section III-B) and (2) **probability learner** that learns the probability distribution of selecting effective mutation methods tailored for each seed group (Section III-C).

Overall Algorithm. We now explain how SeamFuzz works by following Algorithm 2 step by step. SeamFuzz first initializes the set $Crash$ of error-triggering inputs and the set G of seed group information to empty sets at lines 2–3 of Algorithm 2. Then, SeamFuzz selects a seed input s from seed input corpus I. At line 7, the function Cluster identifies the seed group to which the selected seed input s belongs, and returns the corresponding group g, a quintuple $(s_{\mathsf{id}}, S, \mathbf{P}_{\mathsf{id}}, D_{\mathsf{g}}, D_{\mathsf{b}})$. Each component in a quintuple will be discussed in Section III-B and Section III-C. Before sampling mutation methods for a test-case generation, SeamFuzz determines whether to follow the learned probability $\mathbf{P}_{\mathsf{id}}$ (exploit) or the random probability $\mathbf{P}_r$ (explore) at line 11. Since the balance between exploitation and exploration affects greatly on the performance, we, based on trial and error, set the probability for exploitation and exploration to be 70% and 30%, respectively (i.e., the value of ϵ at line 11 is 0.7). SEAMFUZZ follows the selected probability distribution $\mathbf{P}$ to select the effective mutation methods M for mutating the seed input s at line 12, and then it executes the program pgm with the new test-case s' generated by M. For each test-case execution at line 14, SEAMFUZZ accumulates the useful data, where the data is later used to update the probability distribution $\mathbf{P}_{\mathsf{id}}$ optimized for the seed group g in the LEARN function at line 15. By the iterative interaction between CLUSTER and LEARN, SEAMFUZZ is able to learn the sampling probability to choose useful mutation methods optimized for the seed group.

B. Seed Cluster

Given a seed input s and a set G of seed groups, the goal of **seed cluster** is to cluster the selected seed input s into the appropriate seed group with the similar characteristics. To achieve this goal, we define similarity score, denoted $score_{sim}$, and function CLUSTER; the similarity score indicates how close a seed input to a seed group, and the CLUSTER function groups the seed input to the seed group based on the score. To compute the similarity score, we first explain the concept of two similarities: *semantic similarity* and *syntactic similarity*.

Semantic Similarity. Semantic similarity Sim_{sem} describes how similar the seed input behaves to the seed group. More precisely, it depicts the similarity with respect to the execution paths covered by a seed input and a seed group; the more identical covered execution paths, the more similar they are.

In our approach, we express the covered execution path information as a set of covered transitions between two basic blocks, where two transitions with the same basic blocks but different direction are the distinct elements (i.e., $(a \to b) \neq (b \to a)$). Note that the semantic similarity can be computed from other types of execution path information (e.g., branch coverage). As a seed group contains multiple seed inputs, we define the general semantic behaviour of a seed group by computing the set Cov_{all} of all execution path transitions covered by seed inputs in a single seed group. Given a seed group g, let S be the set of seed inputs associated with g (i.e., $g = (\_, S, \_, \_, \_)$). The coverage set Cov_{all} for a seed group g is defined as follows:

$$Cov_{all}(S) = \bigcup_{s \in S} Cov(s) \tag{2}$$

where $Cov(s)$ denotes the set of path transitions covered by the seed input s. The semantic similarity Sim_{sem} of a seed input s to a seed group g is then computed as follows:

$$Sim_{\mathsf{sem}}(s,g) = \frac{|Cov(s) \bigcap Cov_{all}(S)|}{|Cov_{all}(S)|} \tag{3}$$

Intuitively, the semantic similarity is the ratio of the path transitions, which both given seed input s and a set of seed inputs S cover in common, to the ones covered by the set S.

For example, let us assume each set of covered path transitions by s and S is as follows:

$$Cov(s) = \{A, B, D, Z\}, Cov_{all}(S) = \{A, B, C, E, F, K, T, S\}$$

where the capital letters are the symbols for representing each unique transition. We use the capital letters to represent each unique path transition in this paper. Since the two sets have two path transitions (A and B) in common and the size of $Cov_{all}(S)$ is 8, the semantic similarity $Sim_{\mathsf{sem}}(s, g)$ is 0.25.

Syntactic Similarity. Syntactic similarity Sim_{syn} depicts how a seed input is syntactically similar to a seed group. To calculate syntactic similarities between a seed input and a seed group, we consider a seed input s as a bit string with a size of N bits ($s \in \{0, 1\}^N$), and define a representative seed input s_{id} of a seed group for the syntactic characteristics of the seed group. The representative seed input is the one reaching rarely covered path transitions (Cov_{rare}) the most among the seed inputs in the set S, where its role is to depict the syntactical characteristics guiding toward the rarely explored transitions. The set Cov_{rare} will be discussed in more detail in Section III-C.

Given two seed inputs s and s_{id} with sizes of N, we compute the syntactic similarity $Sim_{\mathsf{syn}}(s, g)$ as follows:

$$\begin{aligned} &Sim_{\mathsf{syn}}(s,g) = \tfrac{|\mu(s,s_{\mathsf{id}})|}{N}, g = (s_{\mathsf{id}}, \_, \_, \_, \_), \\ &\mu(s,s') = \{\, 0 \le i < N \mid s[i] = s'[i] \,\}. \end{aligned} \tag{4}$$

where $\mu(s, s_{\mathsf{id}})$ is the set of i-th bit positions with the same value in each seed input, and $s[i]$ indicates the value of the i-th bit of the seed input s.

Like the semantic similarity, the syntactic similarity Sim_{syn} between two seed inputs represents the ratio of the number of the same bit values at the same positions in two seed inputs to the total number of bits in the seed input. If the lengths of the two seed inputs are different, we consider the shorter seed input additionally has the number of different bits as much as the difference in the length; that is, the N always becomes the size of the seed input with longer length.

For example, let us consider three seed inputs as follows:

$$\begin{aligned} s &= \text{“0 1 0 1 0 0 0 0 1 1”}, \\ s' &= \text{“0 1 } \mathbf{1}\ \mathbf{0} \text{ 0 0 0 } \mathbf{1} \text{ 1 1”} \end{aligned} \tag{5}$$

where the bold bits represent the different values at the same positions. The syntactic similarity between s and s' is 0.7 as only three bits are different and the value of N is 10.

Besides two similarities described above, we additionally compute the rareness similarity (Sim_{rare}) for each pair of a seed input and a seed group. The rareness similarity is based on the expectation that the mutation starting from the rarely explored execution transitions would be likely to lead to unexplored deeper program locations.

Hence, we compute an additional score in a case when the seed input covers the infrequently explored transitions which are also covered by the seed group (i.e. $Cov^{group}_{rare}(s, g)$).

$$Cov^{group}_{rare}(s,g) = \{path \in Cov_{rare} | path \in Cov_{all}(\{s\} \cup S)\} \tag{6}$$

With the computed set of rarely-explored transitions covered by both the seed group and the seed input, we calculate the rareness similarity $Sim_{\textsf{rare}}(s, g)$ as follows:

$$Sim_{\textsf{rare}}(s, g) = \frac{\sum_{\textsf{path} \in Cov_{rare}^{group}(s,g)} 1/\textsf{hit}(\textsf{path})}{\sum_{\textsf{path} \in Cov_{rare}} 1/\textsf{hit}(\textsf{path})} \tag{7}$$

where $\textsf{hit}(\textsf{path})$ represents the number of hit counts for the execution path. Intuitively, $Sim_{\textsf{rare}}$ assigns more scores to the given seed input s if it covers more rarely-explored transitions. Unlike the other two similarities, $Sim_{\textsf{rare}}$ has no huge impact on clustering or not, yet it is still useful to identify the most similar seed group when the seed groups have no significance difference between them.

Similarity Score. After obtaining the three different similarities (i.e., syntactic, semantic and rareness), CLUSTER computes the similarity score ($score_{sim}$) between a seed input and each seed group, and determines where the given seed input will belong. Formally, the CLUSTER function is defined as:

$$\textsc{Cluster}(s, G) = \begin{cases} \underset{g \in G}{\operatorname{argmax}}(score_{sim}(s, g)) & \text{if } (d_{\textsf{max}} \geq \gamma) \\ (s, \{s\}, \mathbf{P}_r, \emptyset, \emptyset) & \text{otherwise} \end{cases} \tag{8}$$

where $d_{\textsf{max}}$ represents the highest one among all computed similarity scores, and argmax function returns the seed group having the score $d_{\textsf{max}}$. We obtain the similarity score $score_{sim}$ by using the following equations:

$$\begin{aligned} &score_{sim}(s, g) = Sim(s, g) + Sim_{\textsf{rare}}(s, g), \\ &Sim(s, g) = \alpha \times Sim_{\textsf{sem}}(s, g) + (1 - \alpha) \times Sim_{\textsf{syn}}(s, g) \end{aligned} \tag{9}$$

where the hyper-parameter α mirrors the importance factor of semantic similarity. With trial and error, we set the value of α as 0.9 for our experiments; we demonstrate how the performance of SEAMFUZZ changes with diverse values of α in Section IV-D.

The CLUSTER function calculates the similarity score between a seed input s to each seed group, and returns a seed group with updated set of seed inputs when the highest similarity score $d_{\textsf{max}}$ exceeds the value of γ. If the seed input s is not affiliated to any existing seed groups (i.e., $d_{\textsf{max}}$ is under the value of γ), CLUSTER function organizes a new seed group by initializing the group with the representative seed input s. In this case, we initialize $\mathbf{P}_{\textsf{id}}$ to be uniformly distributed ($\mathbf{P}_r$) as there exists no learned probability distribution for a newly generated group; all group probability distribution $\mathbf{P}_{\textsf{id}}$ starts with a dumb behaviour (i.e., a random probability) and becomes smarter as the learning proceeds.

C. Probability Learner

Now, we describe our **probability learner** which learns probability distributions of selecting effective mutation methods with respect to each seed group.

Sampling Space. Before describing our learning algorithm, we first define a sampling space for selecting the mutation method m that specifies where to mutate (loc) and how to mutate (op). Intuitively, how to mutate is directly related to the mutation operators which are generally predefined, and therefore, the sampling space for choosing mutation operator can be easily determined. The mutation location, however, is unfixed and determined by the given seed inputs of which sizes are diverse, which makes it difficult to define the certain sampling space of choosing mutation location.

To concretely fix the sampling space of choosing loc, we partition the length of the seed inputs by the value of p and make the size of the sampling space for loc to p regardless of the seed inputs. By partitioning the space, we could reduce the sampling space to select where to mutate (loc) from the undefined sizes to the concretely determined size (p). Once our mutation strategy selects a certain mutation partition, it then randomly selects the location within the selected partition. For example, let us assume the value of p is 10, which we also used in our experiments. Given a seed input with size of 40 bits, the size of each partition is 4 bits (e.g., 40 / 10 = 4). If we select 4-th partition by following the learned probability distribution, our seed mutator will randomly select the loc from 13-th bit to 16-th bit which are contained in the 4-th partition of the given seed input.

Thompson Sampling. To learn which mutation methods are effective for better performance, we first note that our problem is naturally expressed as the Multi-Armed Bandit (MAB) problem. The MAB problem assumes N slot machines (bandits), and its goal is to achieve the maximized rewards from them. In each round, we select a bandit to play, and gain rewards by following the probability distribution rigged for each bandit. Intuitively, in our problem setting, each bandit corresponds to each mutation operator op and mutation partition p; in fuzzing technique, the reward may correspond to finding new execution paths or new crash inputs.

Now, the goal of our approach becomes to find the most profitable bandits (i.e., effective mutation methods) to achieve the maximized rewards, where k-th bandit has a probability θ_k of success for the payout reward. To achieve this goal, we employed the Thompson sampling algorithm [1], which is a well-known solution for the classic MAB problem. Thompson sampling builds up a probability model from the observed rewards, and samples the expected value of each bandit from the corresponding model to choose the bandit for the next round. Intuitively, the more rewards we observe for a particular bandit, the more likely we will be to pick that bandit for the next round as the expected value would be the highest among others. If a certain bandit has low probability of success, the probability to select such bandit gets lower as more data is observed.

Our Learning Algorithm. Based on Thompson sampling, we define the probability $\mathbf{P}_i^{\textsf{op}}$ of selecting i-th mutation operator as follows:

$$\mathbf{P}_i^{\textsf{op}} = \frac{\hat{\theta}_i^{\textsf{op}}}{\sum_{0 \leq k < |V^{\textsf{op}}|} \hat{\theta}_k^{\textsf{op}}}, \; \hat{\theta}_i^{\textsf{op}} \sim beta(V_{\textsf{g}}^{\textsf{op}}[i], V_{\textsf{b}}^{\textsf{op}}[i]), \tag{10}$$

where $\hat{\theta}_i^{\textsf{op}}$ represents the expected reward of i-th bandit (i-th mutation operator) sampled from the Beta distribution

$beta(V^{\mathsf{op}}_{\mathsf{g}}[i], V^{\mathsf{op}}_{\mathsf{b}}[i])$. $|V^{\mathsf{op}}|$ represents the number of mutation operators used for mutant generation. A vector $V^{\mathsf{op}}_{\mathsf{g}}[i]$ represents the number of times the i-th mutation operator selected to generate success cases (e.g., interesting test-cases) while $V^{\mathsf{op}}_{\mathsf{b}}[i]$ is for failure cases. Intuitively, we build a probability distribution for mutant operators with sampled expected rewards, which is more likely to choose operators with higher expected rewards. A probability distribution for selecting mutation partition $\mathbf{P}^{\mathsf{p}}$ is obtained in a similar way.

It, however, is problematic to apply the generic Thompson sampling algorithm in our problem setting. Especially, if we naively define the success and failure cases, we could not learn the probability distribution for selecting effective mutation methods. The easiest way to define the success and failure in fuzzing technique is whether the generated test-case is an interesting input; otherwise, we consider the case as a failure.

Unfortunately, this simple classification would lead to an undesirable case in which the probability distribution for selecting effective mutation methods becomes uniformly distributed due to the low efficiency of general grey-box fuzzing techniques [28] as experimentally shown in Section IV-E. More precisely, as the ratio of the number of interesting test-cases (success) to that of uninteresting test-cases (failure) is typically under 0.001%, the likelihood probabilities for all bandit would be also under 0.001. Therefore, the probability distribution for selecting mutation methods becomes uniformly distributed, simulating random sampling. To address this problem, we introduce a specialized classification criteria for success and failure cases.

Classification Criteria. As we already discussed, the naive classification following the metrics of fuzzing technique is too strict for being success and too loose for being failure. Hence, to lower the hurdles for success and to raise the criteria for failure, we further define the following two conditions:

1) s' covers at least one path transition in the Cov_{rare},
2) s' covers over 80% path transitions in the Cov_{common}.

To be a success case, s' needs to satisfy either being interesting or the first condition above. Just being an uninteresting test case is not enough to be a failure case, the second condition must be met. With these two additional conditions, we could raise the number of success and lower the number of failure to properly enjoy the benefits of the Thompson sampling algorithm in our problem setting.

We now define the two sets Cov_{rare} and Cov_{common} of transitions of basic blocks, which are the sets of the rarely-covered and common-covered transitions, respectively. More precisely, we maintain a hit count table for each path transition. This hit count table is updated whenever the program pgm is executed with the generated test-cases. Using this hit count information in ascending order, we build up the two sets Cov_{rare} and Cov_{common} with the top-10% and bottom-30%, which we experimentally set the values for the percentage. These two sets are computed to reflect the latest path coverage information right before CLUSTER function identifies the seed group (line 7 in Algorithm 2).

Based on the classification criteria above, we update our learning data D_{g} and D_{b}. Primarily, the learning data D is a pair of two vectors, mutation operator data (V^{op}) and mutation partition data (V^{p}); i-th element of each vector represents the number of times the i-th mutation operator and partition are selected, respectively. On a basis of the learning data D, D_{g} is a good learning data to accumulate the number of mutation methods used when they are contributed to generate *success test-cases* while D_{b} is for *failure test-cases*.

For example, let us assume that the Cov_{rare}, Cov_{common} and the covered path transitions of two seed inputs (s_1 and s_2) are obtained as follows:

$$\begin{aligned} &Cov_{total} = \{A, B, C, D, E, G, V, W, X, Y, Z\}, \\ &Cov_{rare} = \{A, B, C\}, Cov_{common} = \{V, W, X, Y, Z\} \\ &Cov(s_1) = \{C, X, Y, Z\}, Cov(s_2) = \{D, E, G, V\}, \end{aligned} \tag{11}$$

where Cov_{total} is the set of path transitions covered so far. In this example, both s_1 and s_2 are not interesting test-cases as they failed to cover a new execution path. s_1, however, is considered as a success case as it satisfies the first condition, and we update the good learning data D_{g} with the used mutation methods to generate s_1. For s_2, we do not update either D_{g} or D_{b} since it is not a success and fails to satisfy any of the conditions. Note that if we naively classify the generated test-cases by just checking whether they are interesting or not, both seed inputs would be considered as failure cases.

When comparing the number of interesting test-cases generated in the early and later stage, it is obvious that finding such inputs gets more difficult as the time goes by. In other words, the success rewards earned in the later stage should be relatively better than the one in the early stages, as the success rate is lower in the later stages. We, therefore, gradually increase the rewards to give more profits on the later successes as the fuzzing progresses. We experimentally set the increasing value for reward of success by 10 in every 1 hour, and this will be discussed in Section IV-E.

IV. EXPERIMENTS

In this section, we experimentally evaluate the effectiveness of SEAMFUZZ. We conducted our evaluation to answer the following four research questions.

- **Effectiveness of SEAMFUZZ:** How effectively can our technique improve the performance of fuzzing technique in terms of branch coverage and crash input generation? How does it perform compared to the existing state-of-the art program-adaptive fuzzing technique?
- **Bug-finding ability:** Can our approach find unique bugs that the existing techniques fail to detect?
- **Efficacy of seed clustering:** How does the different granularity of seed clustering affect the performance of fuzzers? How does performance change depending on the different criteria for seed clustering?
- **Effectiveness of learning algorithm:** Is our customized Thompson sampling algorithm essential for improving the performance of our approach? How does the hyper-

TABLE I
BENCHMARK PROGRAMS (AS OF AUGUST 22TH, 2022).

Programs	LoC	Source	Programs	LoC	Source
objdump-2.37	1,654K	[28]	php-parser	1,500K	[17], [30]
arrow-6.0.1	959K	[30]	openssl	695K	[17], [30]
libxml2	451K	[17], [30]	sqlite3	319K	[17], [30]
proj4-9.0.0	279K	[30]	grok-9.7.1	240K	[30]
poppler	196K	[30]	libarchive	164K	[30]
zstd	107K	[30]	infotocap-6.2	83K	[28]
libpng	76K	[30]	podofopdfinfo	62K	[28]

parameters used in learning algorithm affect on the performance of SEAMFUZZ?

A. Evaluation Setting

For evaluation, we used FUZZBENCH [30], a fuzzer benchmarking framework with diverse metrics. We set all parameter values used in FUZZBENCH to their default values, and used the running script file provided by FUZZBENCH. We implemented our technique SEAMFUZZ on top of AFL++ [12], and compared it with two different mutation-based greybox fuzzers AFL++ and AFL++$_{\text{MOPT}}$. AFL++$_{\text{MOPT}}$ is MOPT [28] built on top of AFL++. All three fuzzers we evaluated are based on AFL++ with the version 3.15a. We evaluated each benchmark program during 24 hours with 20 trials to alleviate the influence of the inherent randomness of fuzzing techniques, and reported the average results. All experiments were done on a machine running Ubuntu 20.04 with 64 CPUs and 256GB memory, powered by AMD Ryzen Threadripper 3990X 64-Core Processor.

Benchmarks. Table I shows the benchmark programs and the sources from which we collected the benchmarks. To avoid biasing the evaluation results to a certain benchmark suite, we referred three different sources which were used in the state-of-the-art techniques [44], [12], [28], [46]: FUZZBENCH [30], MAGMA [17], and MOPT [28]. More precisely, we collected 14 programs from a total of 47 programs (13 programs from MOPT, 9 programs from MAGMA and 25 bug-labeled programs from FUZZBENCH).

We used the subset of the programs from those three benchmark sources because evaluating all benchmark programs is very expensive in our evaluation setup. For example, evaluating 14 programs in Table 2 already requires 20,160 hours (24 hours * 20 trials * 14 benchmarks * 3 fuzzers) to complete on a single core. We therefore selected the largest 14 programs among the programs which ran correctly in our evaluation environment. We used the initial seed inputs which the benchmark sources provided for the corresponding benchmark program.

B. Effectiveness of SEAMFUZZ

Table II shows that SEAMFUZZ outperforms two baselines in terms of the number of covered branches and and found crash inputs. In summary, SEAMFUZZ, on average, covered 5.6% and 7.7% more branches than AFL++ and AFL++$_{\text{MOPT}}$, respectively. In terms of crash input generation capability, SEAMFUZZ succeeded to generate 56.4% and 57.1% more crash inputs than AFL++ and AFL++$_{\text{MOPT}}$. Considering each benchmark program, SEAMFUZZ could cover up to 15.9% more branches (e.g., `objdump`) and up to 233.3% more crash inputs (e.g., `php-parser`) than AFL++.

As shown in Table II, SEAMFUZZ generally achieves the highest number of covered branches. For example, on `objdump`, SEAMFUZZ succeeded in covering 5,759 branches while AFL++ and AFL++$_{\text{MOPT}}$ only covered 4,969 and 4,663 branches, respectively. On `php-parser`, however, AFL++$_{\text{MOPT}}$ shows the best performance in terms of the number of covered branches; yet, SEAMFUZZ could still generate about twice more crash inputs than AFL++$_{\text{MOPT}}$. For some programs (e.g., `openssl`), the performance gains of SEAMFUZZ and AFL++$_{\text{MOPT}}$ versus AFL++ are small. We investigated those cases and found that the number of covered branches on those benchmark programs tends to converge to the maximized performance by each fuzzing tool. For example, on the program `openssl`, all fuzzers reached 5,800 branches within 3 hours, where the coverage gains for the rest 18 hours are under 1%.

By computing the Mann-Whitney-p-values provided by FUZZBENCH, we observed that the p-values of SEAMFUZZ against AFL++ and AFL++$_{\text{MOPT}}$ were less than 0.05 for 12 benchmarks, which we could conclude that our results were statistically significant at $\alpha = 0.05$ on those benchmark programs. The exceptional cases were some benchmark programs (e.g., `grok`), where the differences in the number of covered branches between SEAMFUZZ and other fuzzers were small. Also, SEAMFUZZ generally shows the smallest value of the standard deviations compared to other baselines. On the largest 3 programs, for example, the standard deviations for each fuzzer are as follows: (1)AFL++: `objdump`(1,138), `php-parser`(62), `arrow`(42); (2)AFL++$_{\text{MOPT}}$: `objdump`(992), `php-parser`(27), `arrow`(51); (3)SEAMFUZZ: `objdump`(487), `php-parser`(33), `arrow`(39).

Table II also shows that our approach enhances the crash input generation ability. For example, on `proj4`, SEAMFUZZ generated 37.4 crash inputs on average while AFL++ and AFL++ generated 15.6 and 21.3 crash inputs, respectively. Moreover,SEAMFUZZ could generate significantly more crash inputs than the other fuzzers even with small increments of branch coverage. For example, on `php-parser`, SEAMFUZZ generated 233.3% more crash inputs than AFL++ while the gain in the number of covered branches was only 0.4%.

The interesting point is that AFL++ showed generally better performance than AFL++$_{\text{MOPT}}$ in terms of branch coverage. Contrary to the expectation that the program-adaptive approach, MOPT, will always outperform the random approach, the AFL++$_{\text{MOPT}}$ experimentally showed less performance in branch coverage than AFL++ [12], [29]. For example, on all 14 programs, AFL++$_{\text{MOPT}}$ achieved 1.9% less branch coverage and generated 0.4% less crash inputs than AFL++. In bug-finding ability, however, AFL++$_{\text{MOPT}}$ showed its better performance as it detected 87 unique bugs while AFL++ detected 85 bugs as shown in Figure 3.

TABLE II
THE AVERAGE RESULTS FOR 20 TRIALS ON 14 PROGRAMS IN 24 HOURS. LOC: THE NUMBER OF LINES OF CODE FOR ALL SOURCE CODES IN EACH BENCHMARK PROJECT. $Cover$: THE AVERAGE NUMBER OF COVERED BRANCHES. $Crash$: THE AVERAGE NUMBER OF CRASH INPUTS. R_{Cov}: THE RATIO OF THE NUMBER OF COVERED BRANCHES RELATIVE TO AFL++. R_{Crash}: THE RATIO OF THE NUMBER OF CRASH INPUTS RELATIVE TO AFL++.

Program	LoC	AFL++ [12]		AFL++$_{\text{MOpt}}$ [28]				SEAMFUZZ			
		$Cover$	$Crash$	$Cover$	$Crash$	R_{Cov}	R_{Crash}	$Cover$	$Crash$	R_{Cov}	R_{Crash}
objdump	1,653,733	4,969	74.6	4,663	102.6	-6.2%	37.5%	**5,759**	**146.8**	15.9%	96.8%
php-parser	1,500,422	17,010	0.6	**17,161**	0.9	0.9%	50%	17,072	**2.0**	0.4%	233.3%
arrow	958,895	2,430	101.3	2,454	**111.8**	1.0%	10.4%	**2,466**	101.7	1.5%	0.4%
openssl	695,059	5,831	0	5,832	0	0%	0%	**5,843**	0	0.2%	0%
libxml2	450,545	7,608	113.1	6,698	47.8	-12.0%	-57.7%	**7,888**	**176.5**	3.7%	56.1%
sqlite3	318,961	11,581	0	11,222	0	-3.1%	0%	**12,259**	0	5.9%	0%
proj4	279,128	2,572	15.6	2,292	21.3	-10.9%	36.5%	**2,658**	**37.4**	3.4%	139.7%
grok	240,034	5,234	0.0	5,315	0	1.5%	NaN	**5,360**	**21.3**	2.4%	NaN
poppler	195,849	17,436	16.4	18,084	27.6	3.7%	68.3%	**19,588**	**28.2**	12.3%	72.0%
libarchive	164,242	4,781	0.2	4,203	0.3	-12.1%	33.3%	**5,262**	**0.5**	3.7%	150%
zstd	107,194	5,208	0	5,155	0	-1.0%	0%	**5,227**	0	0.4%	0%
libpng	94,561	1,888	0	1,864	0	-1.3%	0%	**1,896**	0	0.4%	0%
infotocap	83,054	1,805	54.1	1,650	60.8	-8.6%	9.4%	**2,033**	**66.2**	12.6%	22.4%
podofopdfinfo	62,000	1,552	11.7	1,563	12.8	0.7%	9.4%	**1,639**	**25.6**	5.6%	118.8%
total	6,803,677	89,905	387.6	88,156	385.9	-1.9%	-0.4%	94,950	606.2	5.6%	56.4%

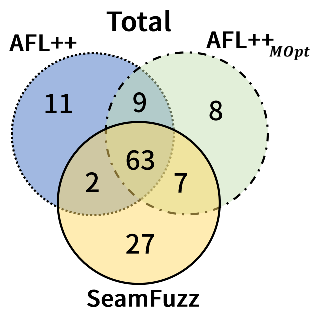

Fig. 3. A Venn diagram for the unique bugs found by each tool.

We additionally checked how the original MOPT and original AFL performed in our evaluation setting. Experimentally, we observed that MOPT showed better performance than AFL; on benchmark programs used in MOPT [28], MOPT covered 2% more branches than AFL while AFL++$_{\text{MOpt}}$ covered 1.9% less branches than AFL++. With the same evaluation setting, the original AFL with our seed-adaptive approach succeeded in reaching 11.4% more branches than AFL. Although AFL++$_{\text{MOpt}}$ showed less performance than MOPT, we chose AFL++ and AFL++$_{\text{MOpt}}$ as our baselines for comparison with our tool SEAMFUZZ. It is because the original AFL and MOPT have not been updated for a long time and AFL++ is an arguably advanced version of AFL with diverse state-of-the-art features [2], [43], [4]. Moreover, AFL++ is actively used in the recent fuzzing papers instead of AFL, nowadays [29], [22], [18].

C. Bug-Finding Ability

Figure 3 shows that our seed-adaptive approach has considerable potential for finding diverse bugs. In summary, SEAMFUZZ could find the largest number of unique bugs (i.e., 99 bugs), including 27 bugs which were not detected by other

TABLE III
FUZZ TARGET PROGRAMS PROVIDED BY MAGMA [17]

Project	Fuzz Target Program
php	exif, json, parser, unserialize
libxml2	read_memory_fuzzer, xmllint
openssl	ans1parse, bignum, client, server, x509
sqlite3	sqlite3_fuzz

baseline fuzzers, while AFL++ and AFL++$_{\text{MOpt}}$ found 85 and 87 unique bugs on the 14 benchmark programs.

To identify each unique bug, we used the bug reports produced by FUZZBENCH. More precisely, we identify each bug as unique one if FUZZBENCH reported different crash type and state. Note that the number of crash inputs, $Crash$, in Table II is different from the number of found bugs; the former represents the average number of all generated crash inputs regardless of the crash type or state for each crash input. That is, numerous inputs may share a single unique bug on the target program. For example, on `grok`, SEAMFUZZ generated 21.3 crash inputs on average, but only two unique heap-buffer-overflow bugs were discovered by these inputs.

SEAMFUZZ found 99 unique bugs in total, and 27 of them are not detected by the other fuzzers AFL++ and AFL++$_{\text{MOpt}}$. SEAMFUZZ detected 9 different types of unique bugs, including null-dereference and heap-use-after-free. Moreover, when considering each benchmark program, SEAMFUZZ was able to detect the largest number of unique bugs as well as the number of the bugs exclusively found by each tool. In particular, on the program `poppler`, SEAMFUZZ could detect 28 out of 38 bugs in total, and 14 bugs were found exclusively by SEAMFUZZ. In contrast, AFL++ and AFL++$_{\text{MOpt}}$ found 16 and 21 bugs, where 3 and 4 bugs were detected solely by the former and the latter, respectively.

TABLE IV
MAGMA[17] BUGS FOUND BY EACH FUZZER. THE TABLE ONLY CONTAINS THE FUZZ TARGET PROGRAMS ON WHICH FUZZERS COULD DETECT BUGS.

Project	Fuzz Target Program	AFL++	AFL++$_{\text{MOpt}}$	SEAMFUZZ
php	exif	PHP004, PHP009, PHP011	PHP004, PHP009, PHP011	PHP004, PHP009, PHP011
libxml2	read_memory_fuzzer	XML001, XML002, XML003, XML009, XML017	XML001, XML003, XML009, XML017	XML001, XML002, XML003, XML009, XML017
openssl	client	SSL002	SSL002	SSL002
	server	SSL002	SSL002, SSL020	SSL002
	x509	SSL009	SSL009	SSL009
sqlite3	sqlite3_fuzz	SQL002, SQL014, SQL015, SQL018, SQL020	SQL002, SQL014, SQL015, SQL018, SQL020	SQL002, SQL014, SQL015, SQL018, SQL020

MAGMA-*labeled bugs.* We additionally evaluated the bug-finding ability of each tool on MAGMA [17], which provides a ground truth of bugs for fuzzing techniques. Since MAGMA provides multiple fuzz target programs (e.g., 2 programs - "read_memory_fuzzer" and "xmllint" - in `libxml2`), we evaluated our experiments on each of these programs. Table III shows the fuzz target programs from MAGMA which we used for the evaluation.

To evaluate bug-injected MAGMA benchmark programs on FUZZBENCH, we inserted assertion statements which specify the bug conditions in MAGMA. By doing so, FUZZBENCH could catch the bugs when the corresponding buggy conditions were satisfied. For example, if MAGMA injects a statement "MAGMA_LOG('bug', a == b)", we insert "assert(a != b)" into the source code; when the buggy condition "a == b" occurs, FUZZBENCH reports the assertion failure as a bug.

Table IV shows the unique MAGMA bugs detected by each fuzzer. More precisely, we conducted experiments for 24 hours with 20 trials on each fuzz target program in Table III, and reported the bugs if the fuzzer detected them at least once during all trials. For example, on `libxml2`, we evaluated all fuzzers on 2 programs in Table III, but the injected MAGMA bugs were detected only when testing "read_memory_fuzzer" as shown in Table IV; on "xmllint", none of the bugs were detected. In summary, all fuzzers could detect the same 14 MAGMA bugs, and SEAMFUZZ and AFL++ detected one more bug "XML002" in `libxml2` while AFL++$_{\text{MOpt}}$ could solely detect "SSL020" in `openssl`.

D. *Efficacy of Seed Clustering*

Table V shows how the performance changes depending on the granularity of seed group. In summary, without our seed clustering algorithm, the performance of the seed-adaptive approach in both path-finding and bug-detection ability significantly decreases.

To investigate the efficacy of seed clustering, we implemented NOCLUSTER and EACHCLUSTER which maintain different number of seed groups. More precisely, NOCLUSTER maintains only a single seed group and clusters all selected seed inputs into this group, which simulates the program-adaptive approach. EACHCLUSTER assigns a seed group per a seed input, which individually applies a different mutation strategy tailored for each seed input. Intuitively, the granularity of seed cluster is determined by the threshold (γ) for seed cluster decision in Section III-B; for example, EACHCLUSTER is exactly the same as SEAMFUZZ of which γ is 1.0. On all 14 programs, NOCLUSTER and EACHCLUSTER covered 4.9% and 4.4% less branches compared to SEAMFUZZ. The performance decrease in crash input generation is more severe; NOCLUSTER and EACHCLUSTER achieved 36.3% and 25.5% less crash inputs, respectively.

TABLE V
PERFORMANCE OF DIFFERENT GRANULARITY OF SEED CLUSTERING.
NOCLUSTER: SEAMFUZZ WITHOUT SEED CLUSTERING.
EACHCLUSTER: SEAMFUZZ WITH ONE SEED CLUSTER PER ONE SEED.

Program	NOCLUSTER		EACHCLUSTER		SEAMFUZZ	
	Cov	*Crash*	*Cov*	*Crash*	*Cov*	*Crash*
objdump	4,567	58.5	5,577	131.0	5,759	146.8
php-parser	16,057	1.3	16,956	0.3	17,072	2.0
arrow	2,290	84	2,398	87.6	2,467	101.7
openssl	5,837	0	5,820	0	5,843	0
libxml2	7,718	146.1	7,599	121.4	7,888	176.5
sqlite3	11,705	0	11,665	0	12,259	0
proj4	2,274	25.1	2,340	22.4	2,658	37.4
grok	5,324	20.3	5,229	0	5,360	21.3
poppler	19,333	21.3	17,879	16.2	19,588	28.2
libarchive	4,788	0.3	4,768	0.3	5,262	0.5
zstd	5,127	0	5,120	0	5,227	0
libpng	1,881	0	1,831	0	1,896	0
infotocap	1,806	24.6	2,016	65.5	2,033	66.2
podofopdfinfo	1,557	4.9	1,568	7	1,639	25.6
total	90,264	386.4	90,766	451.7	94,951	606.2

Interestingly, NOCLUSTER generally showed similar performance to AFL++ which randomly selects mutation methods. Since NOCLUSTER simulates the program-adaptive approach by maintaining only one probability distribution for selecting mutation methods, it could not fully gain the benefits of our learning algorithm specialized to seed-adaptive approach. In contrast, EACHCLUSTER outperformed AFL++ in both branch coverage and crash input generation as it was able to utilize the learned probability distributions for some seed inputs. Since the seed corpus is constantly updated with newly created test cases, and the seed input must have been selected at least once in order to apply the learned probabilities, it is hard to apply the learned probabilities to the seed input for EACHCLUSTER. Moreover, in the same sense, EACHCLUSTER can hardly update the probability distribution with new learning data. As a result, EACHCLUSTER still showed lower performance than SEAMFUZZ which could apply and learn the probabilities more frequently.

We investigated the influence of the syntactic and semantic

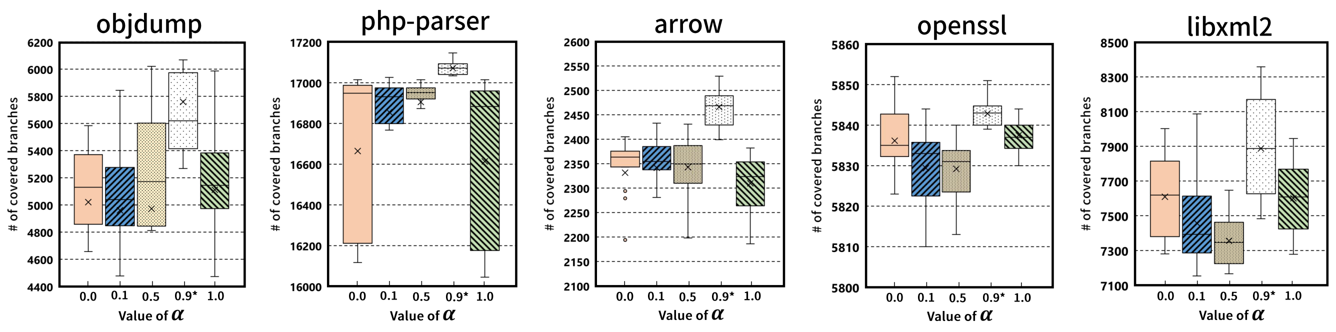

Fig. 4. Performance comparison with the different values of α, an importance factor of semantic similarity, on the largest top-5 programs

TABLE VI
PERFORMANCE WITH DIFFERENT CRITERION FOR DATA CLASSIFICATION. ALL EXPERIMENTS ARE PERFORMED 20 TIMES IN 24 HOURS. NAIVESEAMFUZZ: SEAMFUZZ WITHOUT OUR CLASSIFICATION CRITERIA.

Program	NAIVESEAMFUZZ		SEAMFUZZ			
	Cov	*Crash*	*Cov*	*Crash*	R_{Cov}	R_{Crash}
objdump	4,719	82.9	5,759	146.8	22.0%	77.1%
php-parser	16,955	0.6	17,072	2.0	0.7%	233.3%
arrow	2,376	97.6	2,467	101.7	3.8%	4.2%
openssl	5,830	0	5,843	0	0.2%	0%
libxml2	7,565	116.7	7,888	176.5	4.3%	51.2%
sqlite3	11,501	0	12,259	0	6.6%	0%
proj4	2,313	17.4	2,658	37.4	14.9%	114.9%
grok	4,153	0	5,360	21.3	29.1%	NaN
poppler	18,929	11.9	19,588	28.2	3.5%	137.0%
total	74,341	327.1	78,894	513.9	6.1%	57.1%

TABLE VII
PERFORMANCE DEPENDING ON THE DIFFERENT INCREASING VALUE FOR REWARDS. ALL EXPERIMENTS ARE PERFORMED 20 TIMES IN 24 HOURS. REWARD$_N$: REWARD WITH INCREASING REWARDS BY "N".

Program	REWARD$_0$	REWARD$_1$	REWARD$_5$	REWARD$_{10}$	REWARD$_{20}$
objdump	5,118	5,712	5,773	5,759	5,418
php-parser	16,998	16,977	17,059	17,072	16,967
arrow	2,441	2,303	2,399	2,467	2,374
openssl	5,831	5,824	5,828	5,843	5,834
libxml2	7,481	7,648	7,773	7,888	7,649
sqlite3	11,529	11,606	11,878	12,259	11,500
proj4	2,279	2,544	2,521	2,765	2,226
grok	5,282	5,301	5,283	5,360	4,578
poppler	17,861	19,292	19,422	19,588	18,965
total	74,820	77,207	77,936	78,894	74,611

similarities on the performance of SEAMFUZZ. To do so, we set the value of α, an importance factor of semantic similarity, in four different values, and selected the top-5 largest programs for evaluation. Intuitively, SEAMFUZZ only considers the syntactic similarities when the value of α is 0.0; in contrast, it only uses the semantic similarities for seed clustering if α is equal to 1.0. As shown in Figure 4, SEAMFUZZ of which the value of α is denoted with a star outperforms the other settings. It achieved the highest median and average values as well as the narrow range of interquartile; that is, considering both syntactic and semantic similarities in proper proportions is critical for the performance of SEAMFUZZ.

We also examined how many unique seed inputs are clustered into each seed group as well as the number of maintained groups. In summary, the average number of maintained seed groups and the total seed inputs selected for seed mutation on 14 programs is 398 and 1,663, respectively.

E. Effectiveness of Learning Algorithm

We evaluated how our classification criteria for success and failure cases in Thompson sampling algorithm affects the overall performance of SEAMFUZZ on the largest top 9 programs. More precisely, we compared SEAMFUZZ to NAIVESEAMFUZZ which does not utilize our criteria but identifies the success and failure based only on the interesting test-cases in Section III-C. In summary, our customized criteria for Thompson sampling improves both the path-discovering and crash-generation abilities; it helps to cover 6.1% more branches and generate 57.1% more crash inputs compared to the naive classification. Interestingly, the general performance of NAIVESEAMFUZZ is similar to that of AFL++ in Table II. It is because the learned probability of NAIVESEAMFUZZ generally follows the uniform distribution due to the enormous number of failure cases as discussed in Section III-C.

We investigated how the increasing value of the success reward affects the performance of our approach. To observe the effect of dynamic changes in reward on performance, we evaluated SEAMFUZZ with five different settings; REWARD$_N$ increases the rewards by "N". As shown in Table VII, dynamically updating rewards with proper values can significantly improve the performance of our approach. For example, REWARD$_0$ that does not dynamically update the rewards shows similar performance with AFL++ while REWARD$_{10}$ which is the default SEAMFUZZ performs the best on the 9 programs. Interestingly, updating with large values degrades the performance as shown by REWARD$_{20}$. We additionally checked that the cost for learning process, including seed cluster, depends on target programs. For example, the learning process takes 9.4% of total time testing `objdump` on average.

F. Threats to Validity

(1) We implemented our approach on top of AFL++ and evaluated it for AFL++ and AFL++$_{\text{MOPT}}$. We chose them as AFL++ is actively maintained and updated with diverse state-of-the-art techniques. The results reported in this paper may not be valid for other base fuzzer, such as LIBFUZZER [37]. (2) Our approach involves some hyper-parameters, e.g., α, which were selected with trial and error. These values, however, may not be the best ones as we could not evaluate all

possible combinations for hyper-parameters. (3) We evaluated our approach on 14 large programs collected from various sources [30], [17], [28] with 20 trials in 24 hours. However, these programs might not be representative enough.

V. Related Works

In this section, we discuss prior works related to our work.

1) **Mutation-based greybox fuzzing:** Mutation-based greybox fuzzing has been extensively studied in recent years, e.g., [5], [9], [31], [37], [40]. AFL [49], for example, is perhaps the most widely-used mutation-based fuzzing tool due to its ease of use and impressive performance. Based on AFL, many descendants [3], [12], [13], [48] have been emerged with diverse techniques to boost its performance. Along with the descendants of AFL, and other mutation-based fuzzing techniques, we can classify them into two categories based on what they focus on: seed mutation and seed selection.

First, numerous researches have been conducted focusing on the seed mutation strategy to achieve better performance [6], [10], [23], [28], [45], [46]. FairFuzz [23], for example, guides the seed mutation strategy to mutate the desired locations of the seed inputs by masking the locations not to mutate. By employing a particle swarm optimization algorithm, MOpt [28], learns the optimal probability distribution of choosing mutation operators for a target program. CMFuzz [46] improves the mutation strategy by resolving the randomness of choosing mutation operators by considering the input file formats. These existing techniques learn the optimal mutation strategies with regard to only the target program. Our focus, however, is on learning effective mutation strategies optimized for each seed group.

Second, as seed selection strategy is also known as a critical component or mutation-based fuzzing technique, several researches have focused on how to select a promising seed input to fuzz next [4], [19], [21], [26], [32], [34], [44], [29]. By using a Markov chain, AFLFast [4] prioritizes the seed inputs with low frequency paths to guide the fuzzer explore the rarely covered paths more. With introducing a concept of input potential, Cerebro [26] measures the complexity of uncovered code locations, and guides fuzzer to select the seed inputs which are likely to penetrate such location. VUzzer [34] prioritizes the inputs with hard-to-reach paths by modeling execution paths with control-flow features. AFL-HIER [44], with multi-leveled coverage metric, improves the seed selection strategy by allocating energy on the clusters of seed inputs which are likely to cover new program paths. Although ours is orthogonal to these techniques, we expect that combining ours with them would increase the overall performance.

Several researches have focused on improving the bug finding ability of mutation-based fuzzing technique [7], [8], [11], [16], [25], [27], [31]. UAFL [42], for instance, models use-after-free vulnerability as typestate properties and uses static analysis for discovering the vulnerabilities. With static analysis on the target program and target sites, Hawkeye [8] collects information which is later used for seed selection and seed mutation. Our proposal of seed-adaptive mutation strategy is largely orthogonal to these works. Also, UAFL and Hawkeye leverage source code to run static analysis, which is sometimes unavailable in particular for commercial, off-the-shelf (COTS) software.

2) **Machine learning based fuzzing:** Recently, machine learning technique has emerged as a trending technology that brings a new paradigm to many fields and improves the overall performance of other technologies. Along with this trend, many fuzzing techniques have been proposed to boost the performance of fuzzing with the help of machine learning [5], [15], [24], [33], [38], [39], [44]. MTFuzz [38], for example, uses a multi-task neural network to learn a compact embedding of the input space for effective seed mutation guidance. Speed-Neuzz [24] leverages neural networks to model the branch behaviours of the testing program for locating the critical bytes in the seed input.

Using the reinforcement learning algorithm, several researches on fuzzing techniques have modeled the problem as Multi-Armed Bandit problem [36], [20]. BanditFuzz [36] firstly presented a reinforcement learning algorithm for fuzzing SMT solver, which leverages the Thompson sampling algorithm. Siddharth Karamcheti et al. [20] leverages the Thompson sampling algorithm to fine-tune the mutator distribution adaptively to an individual program. Our work broadly lies in this line of research; to our knowledge, our work is the first to use a learning algorithm for selecting effective mutation methods optimized for each seed group.

VI. Conclusion

Recently, program-adaptive mutation strategies for greybox fuzzers have successfully advanced existing program-agnostic fuzzers. In this paper, we took one step further in this direction, calling for attention to the transition from the program-adaptive to seed-adaptive approaches. Our key idea is to cluster the seed inputs into seed groups based on syntactic and semantic similarity and learn mutation strategies optimized for each seed group online using a variant of Thompson sampling. We demonstrated that our seed-adaptive approach greatly enhances the state-of-the-art program-adaptive technique in terms of coverage and bug-finding.

Acknowledgement

This work was supported by Samsung Electronics Co., Ltd (IO201216-08209-01), Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2020-0-01337,(SW STAR LAB) Research on Highly-Practical Automated Software Repair), the MSIT(Ministry of Science and ICT), Korea, under the ICT Creative Consilience program(IITP-2023-2020-0-01819) supervised by the IITP(Institute for Information & communications Technology Planning & Evaluation), and the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT)(No. 2021R1A5A1021944, No. NRF-2021R1C1C2006410, No. RS-2023-00208094).

REFERENCES

[1] Shipra Agrawal and Navin Goyal. Analysis of thompson sampling for the multi-armed bandit problem. *Journal of Machine Learning Research*, 23, 11 2011.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[3] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. *Directed Greybox Fuzzing*. Association for Computing Machinery, New York, NY, USA, 2017.

[4] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2019.

[5] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. Deep reinforcement fuzzing. 01 2018.

[6] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *2015 IEEE Symposium on Security and Privacy*, pages 725–741, 2015.

[7] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. MUZZ: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2325–2342. USENIX Association, August 2020.

[8] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2095–2108, New York, NY, USA, 2018. Association for Computing Machinery.

[9] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018.

[10] Jiale Deng, Xiaogang Zhu, Xi Xiao, Sheng Wen, Qing Li, and Shutao Xia. Fuzzing with optimized grammar-aware mutation strategies. *IEEE Access*, 9:95061–95071, 2021.

[11] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. The use of likely invariants as feedback for fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021.

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

[13] Yu Fu, Siming Tong, Xiangyu Guo, Liang Cheng, Yang Zhang, and Dengguo Feng. Improving the effectiveness of grey-box fuzzing by extracting program information. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 434–441, 2020.

[14] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696, 2018.

[15] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, page 50–59. IEEE Press, 2017.

[16] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, page 49–64, USA, 2013. USENIX Association.

[17] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), nov 2020.

[18] Grant Hernandez, Marius Muench, Dominik Maier, Alyssa Milburn, Shinjo Park, Tobias Scharnowski, Tyler Tucker, Patrick Traynor, and Kevin Butler. Firmwire: Transparent dynamic analysis for cellular baseband firmware. 01 2022.

[19] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. *Seed Selection for Successful Fuzzing*, page 230–243. Association for Computing Machinery, New York, NY, USA, 2021.

[20] Siddharth Karamcheti, Gideon Mann, and David S. Rosenberg. Adaptive grey-box fuzz-testing with thompson sampling. *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*, 2018.

[21] Sungjin Kim, Jaeik Cho, Changhoon Lee, and Taeshik Shon. Smart seed selection-based effective black box fuzzing for iiot protocol. *The Journal of Supercomputing*, pages 1–15, 2020.

[22] Ahcheong Lee, Irfan Ariq, Yunho Kim, and Moonzoo Kim. Power: Program option-aware fuzzer for high bug detection ability. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 220–231, 2022.

[23] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. New York, NY, USA, 2018. Association for Computing Machinery.

[24] Yi Li, Xi Xiao, Xiaogang Zhu, Xiao Chen, Sheng Wen, and Bin Zhang. Speedneuzz: Speed up neural program approximation with neighbor edge knowledge. In *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 450–457, 2020.

[25] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 627–637, New York, NY, USA, 2017. Association for Computing Machinery.

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

[27] Yuwei Li, Shouling Ji, Chenyang Lyu, Yuan Chen, Jianhai Chen, Qinchen Gu, Chunming Wu, and Raheem Beyah. V-fuzz: Vulnerability prediction-assisted evolutionary fuzzing for binary programs. *IEEE Transactions on Cybernetics*, pages 1–12, 2020.

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

[29] Chenyang Lyu, Hong Liang, Shouling Ji, Xuhong Zhang, Binbin Zhao, Meng Han, Yun Li, Zhe Wang, Wenhai Wang, and Raheem Beyah. Slime: Program-sensitive energy allocation for fuzzing. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 365–377, New York, NY, USA, 2022. Association for Computing Machinery.

[30] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. Fuzzbench: An open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1393–1403, New York, NY, USA, 2021. Association for Computing Machinery.

[31] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. Sfuzz: An efficient adaptive fuzzer for solidity smart contracts. New York, NY, USA, 2020. Association for Computing Machinery.

[32] Shankara Pailoor, Andrew Aday, and Suman Jana. Moonshine: Optimizing os fuzzer seed selection with trace distillation. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, page 729–743, USA, 2018. USENIX Association.

[33] Mohit Rajpal, William Blum, and Rishabh Singh. Not all bytes are equal: Neural byte sieve for fuzzing. 11 2017.

[34] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. *VUzzer : Application - aware Evolutionary Fuzzing*. NDSS'17. 2017.

[35] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 861–875, San Diego, CA, August 2014. USENIX Association.

[36] Joseph Scott, Trishal Sudula, Hammad Rehman, Federico Mora, and Vijay Ganesh. Banditfuzz: Fuzzing smt solvers with multi-agent reinforcement learning. In *Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings*, page 103–121, Berlin, Heidelberg, 2021. Springer-Verlag.

[37] Kosta Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 157–157, 2016.

[38] Dongdong She, Rahul Krishna, Lu Yan, Suman Jana, and Baishakhi Ray. Mtfuzz: Fuzzing with a multi-task neural network. In *Proceedings of the*

28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020, page 737–749, New York, NY, USA, 2020. Association for Computing Machinery.

[39] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program smoothing. pages 803–817, 05 2019.

[40] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Krügel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, 2016.

[41] R. Swiecki. Honggfuzz, 2014. https://github.com/google/honggfuzz.

[42] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 999–1010, 2020.

[43] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *RAID*, 2019.

[44] Jinghan Wang, Chengyu Song, and Heng Yin. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. In *NDSS*, 2021.

[45] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735, 2019.

[46] Xiajing Wang, Changzhen Hu, Rui Ma, Donghai Tian, and Jinyuan He. Cmfuzz: context-aware adaptive mutation for fuzzers. *Empirical Software Engineering*, 26, 01 2021.

[47] Tai Yue, Yong Tang, Bo Yu, Pengfei Wang, and Enze Wang. Learnafl: Greybox fuzzing with knowledge enhancement. *IEEE Access*, 7:117029–117043, 2019.

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

[49] Michal Zalewski. American fuzzy lop, 2014.