On Interaction Effects in Greybox Fuzzing

Konstantinos Kitsios
konstantinos.kitsios@uzh.ch
University of Zurich
Zurich, Switzerland

Marcel Böhme
marcel.boehme@acm.org
Max Planck Institute for Security and Privacy
Bochum, Germany

Alberto Bacchelli
University of Zurich
Zurich, Switzerland
bacchelli@ifi.uzh.ch

ABSTRACT

A greybox fuzzer is an automated software testing tool that generates new test inputs by applying randomly chosen mutators (e.g., flipping a bit or deleting a block of bytes) to a seed input in random order and adds all coverage-increasing inputs to the corpus of seeds. We hypothesize that the *order* in which mutators are applied to a seed input has an impact on the effectiveness of greybox fuzzers. In our experiments, we fit a linear model to a dataset that contains the effectiveness of all possible mutator pairs and indeed observe the conjectured interaction effect. This points us to more efficient fuzzing by choosing the most promising mutator sequence with a higher likelihood.

We propose MuoFuzz, a greybox fuzzer that learns and chooses the most promising mutator sequences. MuoFuzz learns the conditional probability that the next mutator will yield an interesting input, given the previously selected mutator. Then, it samples from the learned probability using a random walk to generate mutator sequences. We compare the performance of MuoFuzz to AFL++, which uses a fixed selection probability, and MOPT, which optimizes the selection probability of each mutator in isolation. Experimental results on the FuzzBench and MAGMA benchmarks show that MuoFuzz achieves the highest code coverage and finds four bugs missed by AFL++ and one missed by both AFL++ and MOPT.
Data and material: https://doi.org/10.5281/zenodo.17391100

CCS CONCEPTS

• **Security and privacy → Software security engineering**; • **Software and its engineering → Software testing and debugging**.

KEYWORDS

software security, fuzzing, mutation strategy

ACM Reference Format:
Konstantinos Kitsios, Marcel Böhme, and Alberto Bacchelli. 2025. On Interaction Effects in Greybox Fuzzing. In *Proceedings of 48th IEEE/ACM International Conference on Software Engineering (ICSE 2026).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE 2026, April 12–18, 2026, Rio de Janeiro, Brazil
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN XXX-X-XXXX-XXXX-X/XX/XX
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Mutation-based greybox fuzzing aims to automatically detect unexpected software behaviour [5, 6, 20, 27, 28, 48]. Given a target program and a corpus of seed inputs (or simply *seeds*) to that program, fuzzers select one seed and mutate it to produce a mutated input (or simply *input*), which they feed to the target program and gather code coverage feedback. If the input covers new code, it is added to the seed corpus for further mutations. By repeating this process for millions of inputs, fuzzers automatically reach and test deep program locations.

AFL [47] and its more recent successor, AFL++ [11], are the most widely used fuzzers. To mutate a seed, AFL++ relies on 32 mutators, ranging from simple random bit flipping to disruptive deletions of byte blocks from the seed. Many of these mutators are applied consecutively to the seed (forming a *mutator sequence*) and the algorithm that controls how mutator sequences are generated is called *mutation strategy*. The mutation strategy of AFL++ is straightforward: each of the 32 mutators has a fixed probability of being selected. Researchers [21, 25, 29, 46] have proposed mutation strategies that adjust the probability of each mutator, such that mutators that produce more coverage-increasing inputs (also called *interesting* inputs) have a higher probability of being selected.

The aforementioned studies model and leverage the probability that each mutator will yield an interesting input in isolation, and then sample l mutators from this probability to create a mutator sequence of length l. This process assumes that each mutator in the sequence is independent of the others: the probability of selecting the next mutator does not take into account the already chosen mutators.

We hypothesize that there may exist an interaction effect between mutators, which can be leveraged to further increase the number of interesting inputs and, in turn, the fuzzer's performance. Based on this hypothesis, we propose a threefold contribution:

- Empirical evidence that some mutators, when combined, produce more interesting inputs than others.
- A mutation strategy, implemented into MuoFuzz [19], that leverages our first finding by sampling the next mutator from a probability distribution conditioned on the previously selected mutator.
- Empirical evidence on the effectiveness of MuoFuzz by comparing against state-of-the-art fuzzers in terms of achieved code coverage and found bugs.

For our first contribution, we fit a linear model where the two independent variables are the two mutators in a sequence of length two and the dependent variable is the number of interesting inputs produced by this sequence. We collect our dataset by running AFL++ in 13 target programs. By running an Analysis of Variance (ANOVA) [12] on the fitted model, we find that the interaction term

explains a statistically significant proportion of the variance. This means that the interaction effect between two mutators affects the number of interesting inputs these mutators produce.

For our second contribution, we leverage this newly found interaction effect: We propose a strategy for generating mutator sequences where the probability of selecting the next mutator is conditioned on the previously selected mutator. We implement this strategy into MuoFuzz (**Mu**t + Du**o** + **Fuzz**), which works in two phases: During the (1) *training phase*, MuoFuzz learns the conditional probability that a mutator will yield an interesting input given the previous mutator in the sequence; during the (2) *guided mutation phase*, MuoFuzz samples from the learned probability using a random walk [33].

For our final contribution, we compare the performance of MuoFuzz with AFL++, which uses a fixed selection probability, and MOPT [29], which optimizes the selection probability of each mutator in isolation, without considering the interaction effect between mutators. MuoFuzz achieves the highest code coverage in 10 out of 13 FuzzBench [31] programs. Moreover, it finds four bugs that AFL++ missed, as well as one bug that both AFL++ and MOPT missed in the MAGMA benchmark [16].

2 BACKGROUND

In this section, we present the inner workings of mutation-based greybox fuzzers to provide the necessary background for the rest of the paper.

Algorithm 1: Mutation-based greybox fuzzing. The part we modify is highlighted.

```
Input  : Target program p, seed corpus S, set of mutators M
Output : Corpus with crashing inputs Crash
Crash ← ∅;
repeat
    s ← SelectSeed(S);
    for i_input = 1 to num_inputs_for_this_seed do
        M ← ∅;
        for n = 1 to l do
            m_n ~ Pr(M); // fixed for AFL++; learned for MOPT;
                         we propose Pr(M | m_{n-1}).
            M ← M ∪ {m_n};
        end
        s' ← Mutate(s, M);
        Crash, S ← Execute(p, s', Crash, S);
    end
until timeout reached;
return Crash;
```

2.1 Mutation-based Greybox Fuzzing

A mutation-based greybox fuzzer takes as input a target program and a corpus of seed inputs. These initial seeds are usually human-written inputs aiming to provide a good starting point for mutation. The fuzzer automatically generates inputs for the target program by following the steps of Algorithm 1. First, it selects a seed from the corpus to mutate (line 3). The probability of selecting a seed depends on heuristic rules: For example, seeds that have been selected in the past have a lower probability, while seeds that are smaller in size have a higher probability. Then, the fuzzer decides the number of mutated inputs that will be produced from this seed (loop limit in line 4). This number ranges from 16 to many thousands and depends on similar heuristics. Afterward, the fuzzer mutates the seed to generate an input, following the mutation strategy described in the next section (Section 2.2). Finally, the fuzzer feeds the input to the program (line 11) and monitors its behaviour: If the input crashes the program, then it is considered a *potential* bug and is returned for human inspection (line 14). If the input achieves new code coverage, it is considered *interesting* and is added to the seed corpus to be further mutated in future iterations [48].

Table 1: AFL++ mutators and their selection probability.

ID	Description	Type	Probability
1	flip a random bit	unit	0.043
2	replace a random byte with an interesting value	unit	0.035
3	replace two adjacent bytes with interesting values	unit	0.023
4	replace two adjacent bytes with interesting values (be*)	unit	
5	replace four adjacent bytes with interesting values	unit	
6	replace four adjacent bytes with interesting values (be)	unit	
7	subtract a value between 1 and 35 from a random byte	unit	
8	add a value between 1 and 35 to a random byte	unit	
9	subtract a value between 1 and 35 from two adjacent bytes	unit	
10	subtract a value between 1 and 35 from two adjacent bytes (be)	unit	
11	add a value between 1 and 35 to two adjacent bytes	unit	
12	add a value between 1 and 35 to two adjacent bytes (be)	unit	
13	subtract a value between 1 and 35 from four adjacent bytes	unit	
14	subtract a value between 1 and 35 from four adjacent bytes (be)	unit	
15	add a value between 1 and 35 to four adjacent bytes	unit	
16	add a value between 1 and 35 to four adjacent bytes (be)	unit	
17	set a random byte to a random value	unit	
18	increase a random byte by 1	unit	
19	decrease a random byte by 1	unit	
20	flip all the bits of a random byte	unit	
21	swap a block of bytes between two positions in the seed	chunk	
22	delete a block of bytes	chunk	
23	overwrite a block of the seed with a dictionary entry	chunk	
24	insert a dictionary entry into a random position of the seed	chunk	
25	overwrite a block of the seed with an auto-dictionary entry	chunk	
26	insert an auto-dictionary entry into a random position	chunk	
27	select a block from another seed of the corpus, and use it to overwrite a block of the seed	chunk	
28	select a block from another seed of the corpus and insert it into a random position of the seed	chunk	
29	select a block and insert a copy of it at a different position	chunk	
30	insert a block of constant bytes to a random position in the seed. The constant block can either be a random value or a part of the original seed.	chunk	
31	select a block and overwrite another block with it	chunk	0.039
32	select a block and overwrite it with a fixed byte value, which can be either a random byte or a byte from the original seed	chunk	0.020

*be = big endian

2.2 Mutation Strategy

The mutation strategy of AFL++, on top of which we develop our fuzzer, consists of three stages. The selected seed is first propagated through the *deterministic stage* where a set of deterministic mutations are applied. For example, all the bits of the seed are flipped, one at a time. The deterministic stage targets "low-hanging fruits" and is not effective in deep program locations due to its simplicity.

Then, the seed goes through the *havoc stage*, which is the most effective of the three stages [46]. The havoc stage comes with 32 predefined mutators shown in Table 1. We categorize the mutators into *unit* and *chunk* mutators, following previous work [46]. Unit mutators perform lightweight modifications to the seed, such as flipping a random bit. Chunk mutators, on the other hand, disruptively modify the seed, for example by deleting a whole block. AFL++ performs the following steps in the havoc stage. First, it selects the mutator sequence length l (loop limit in line 6), i.e., how many stacked mutators to apply to the seed. This number ranges from 2 to 16, with lower values having higher probability. Then,

it samples from the set of mutators l times using the predefined probabilities of Table 1, creating a sequence of l mutators (lines 7–8). The mutator sequence is then applied to the seed sequentially to produce the mutated input (line 10). The input is finally fed into the target program and coverage feedback is collected as described in Section 2.1.

The third stage of the AFL++ mutation strategy is the *splice stage*. It disruptively mutates the seed by selecting a random block and concatenating it with a random block from a different seed. Both the splice stage and the deterministic stage have less impact than the havoc stage [46], thus this work, similar to previous works [25], focuses on the havoc stage. For the rest of this paper, the term mutation strategy will refer to the havoc stage of the mutation strategy.

3 RELATED WORK

We present mutation strategies proposed by previous studies to improve the default mutation strategy of AFL++, which is program-agnostic: it assigns a fixed, predefined probability to each of the 32 mutators, regardless of the target program it tests. This approach is suboptimal since some mutators could work better in some programs and worse in others. This led researchers to investigate mutation strategies that adjust the probability of each mutator based on how many interesting inputs the mutator produces.

MOPT [29] is one of the most successful program-adaptive mutation strategies [37] and is included in the official AFL++ repository. It continuously adjusts the mutator probabilities using a genetic algorithm inspired by the Particle Swarm Optimization technique, which aims to find optimal selection probabilities for the 32 mutators so that they maximize the likelihood of generating an interesting input. SLOPT [21] sets a constraint on the mutation strategy of AFL++: only a single mutator will be used to create a mutator sequence. This simplification reduces the problem to the subproblems of (1) what mutator will be used in each sequence and (2) how many times to stack the mutator. SLOPT views these two problems as multi-armed bandit (MAB) problems and uses the UCB algorithm [2] to solve them. $\text{HAVOC}_{\text{MAB}}$ [46] treats the problems of selecting the optimal length of the mutator sequence l, as well as each mutator in the sequence, as two MAB problems, and uses established MAB algorithms to solve them. SeamFuzz [25] proposes a seed-adaptive mutation strategy, where different mutator selection probabilities are used for different seeds. Seeds are clustered according to their similarity, and one selection probability is learned for each cluster. CMFuzz [45] encodes the seeds as context and uses contextual MAB algorithms to solve it, leading to different mutator probabilities for different seeds. VUzzer [36] has two mutators and selects them with a fixed probability, while focusing on optimizing the location of the seeds to mutate. MendelFuzz [49] evaluates and improves the deterministic mutation stage, which involves applying simple mutators (e.g., byte flipping) in all the possible seed positions. Finally, HonggFuzz [41], similar to AFL++, implements a program-agnostic mutation strategy with slightly different mutators than AFL++.

All of these studies generate mutator sequences by considering each mutator in the sequence as independent from the others. This assumption allows the use of MAB algorithms, where, by definition, each arm is independent from the others [24]. Our work focuses on selecting the optimal mutator conditioned on the previously selected one, which makes it orthogonal to previous work.

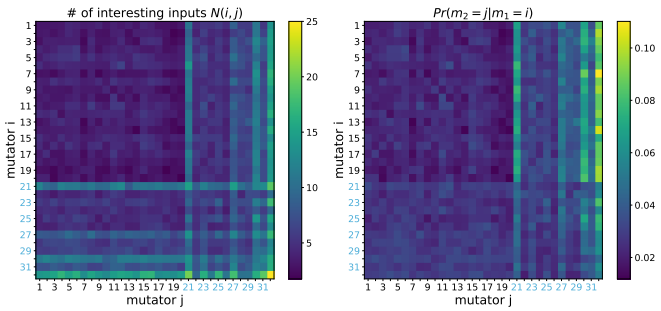

Figure 1: The left figure shows the number of interesting inputs generated by the mutator sequence $\langle i, j \rangle$, where i is the mutator in the row and j is the mutator in the column. The conditional probability on the right figure is derived by dividing each row of the left figure by its sum. As such, each row i of the right matrix is a probability distribution, denoting how probable is that selecting j as the second mutator will yield an interesting input, given that the first mutator is i. These concern the `freetype` target program.

4 INVESTIGATING THE INTERACTION EFFECT BETWEEN MUTATORS

We hypothesize that some mutators, when combined, are more effective than others and we experimentally test our hypothesis.

> **RQ1.** Can we measure an interaction effect between two mutators in a mutator sequence?

To test the existence of the interaction effect, we collect a dataset that captures the effectiveness (i.e., number of interesting inputs discovered) of each possible pair of mutators. We note that other factors, like mutator position, can contribute to the number of interesting inputs discovered by a mutator pair. However, since fuzzers perform hundreds of millions of mutations, it is reasonable to assume that these factors even out across all mutator pairs.

Then, we fit a linear model with an interaction term to the collected dataset and conduct a two-way ANOVA [12] to test the significance of the model's interaction term. To collect the dataset, we run AFL++ in 13 target programs following previous work [6, 18, 25, 29, 46], as it is the most widely used fuzzer. However, our methodology is applicable to sets of mutators other than AFL++, and we expect our findings to generalize to other mutation-based greybox fuzzers like libFuzzer [15] that have a similar set of mutators. The details of the dataset collection process are provided in Section 4.1, followed by the model fitting in Section 4.2 and the results in Section 4.3.

4.1 Dataset Collection

Let $\mathcal{M} = \{1, 2, ..., M\}$ be the IDs of the available mutators and let $\langle m_1, m_2, ..., m_l \rangle \in \mathcal{M}^l$ denote a mutator sequence of length l. To answer our first research question, we need a dataset containing the number of interesting inputs generated by each possible pair of mutators $\langle i, j \rangle \in \mathcal{M}^2$. To collect this dataset, we first select 13 target programs that were used in previous work [25, 35]. The programs are part of the FuzzBench [31] benchmark and are shown in the first column of Table 3. More details on their selection are provided in Section 6.2. We study each program individually, since some mutator sequences may be efficient in some programs but not in others. Our goal is to generate data points $N(i, j), i, j \in \mathcal{M}$ for each program that contain the number of interesting inputs generated by the mutator sequence $\langle i, j \rangle$. To do this, we fuzz the selected target programs using AFL++ with three modifications:

M1: We limit the length of the mutator sequences to $l = 2$ to study the interaction effect between 2 mutators. Studying the interaction effect of $l > 2$ mutators is computationally challenging, as we show in Section 5.4.3, where we study the interaction effect of triplets by using $l = 3$. We observe that, due to combinatorial explosion, most triplets remain unobserved during training because they do not produce a coverage-increasing input.

M2: We sample the mutators uniformly at random instead of following the default probabilities of AFL++, to gather the same amount of data for each mutator pair.

M3: We maintain a $|\mathcal{M}| \times |\mathcal{M}|$ matrix $N(i, j)$ that holds the number of interesting inputs produced by the sequence $\langle i, j \rangle$. Each time $\langle i, j \rangle$ generates an interesting input, we increase $N(i, j)$ by 1.

We fuzz each target program with AFL++ for 20 trials, hence we obtain one matrix for each trial $k \in \{1, 2, ..., 20\}$, which we call $N(i, j)^{(k)}$. We experiment with different values for the length T of the fuzzing campaigns, namely $T \in \{1, 5, 24\}$ hours. We find that the significance of the interaction effect does not change across the three values of T because AFL++ discovers most of the interesting inputs within the first hours of fuzzing. Figure 1 (left) shows the collected data $N(i, j)$ averaged over the trials k for the `freetype` program, while the plots for the rest of the programs are available in our replication package [19].

4.2 Model Fitting

We use the dataset $N(i, j)^{(k)}$ to train a linear model that, given two mutators $i, j \in \mathcal{M}$, will predict the expected number of interesting inputs generated by the sequence $\langle i, j \rangle$ in a fuzzing campaign for the given target program. We selected a linear model for simplicity and interpretability. Given the high goodness-of-fit in Table 2, we did not try higher-order models. The linear model has categorical independent variables (both i and j are categorical) and a numerical dependent variable [9], so it is of the form:

$$\hat{N}(i, j) = \mu + \alpha_i + \beta_j + \gamma_{ij} \quad (1)$$

where μ is the constant bias, α_i is the effect of the first mutator in the sequence, β_j is the effect of the second mutator in the sequence, and γ_{ij} is the interaction effect between the two mutators.

The intuition is that the mutator i in the sequence $\langle i, j \rangle$ is associated with a constant effect α_i (or β_j for mutator j). The effect a_i means that, whenever i is the first mutator in the sequence, we should expect an increase (or decrease) in the number of interesting inputs (relative to the bias μ) of α_i inputs. The same holds for β_j. Regarding γ_{ij}, it denotes the interaction effect of mutators i and j and is interpreted as follows: Suppose that we expect the mutator i in the first position of the sequence (independent of the second) to increase the number of interesting inputs by α_i and the mutator j in the second position of the sequence (independent of the first) to increase the number of interesting inputs by β_j. Now, suppose that when i is combined with j we observe a significant increase (or decrease) by $\gamma_{ij} \neq 0$; this would mean that there exists an interaction effect between i and j. If, on the contrary, $\gamma_{ij} = 0$, it would mean that there is no interaction effect.

Table 2: The linear model with the interaction term ($\gamma_{ij} \neq 0$) has higher goodness-of-fit than the model without the interaction term ($\gamma_{ij} = 0$).

	R^2			R^2_{adj}		
Target	$\gamma_{ij} = 0$	$\gamma_{ij} \neq 0$	δ	$\gamma_{ij} = 0$	$\gamma_{ij} \neq 0$	δ
proj4	0.758	**0.855**	0.097	0.757	**0.847**	0.090
json	0.629	**0.853**	0.224	0.628	**0.848**	0.220
curl	0.672	**0.820**	0.148	0.672	**0.814**	0.142
re2	0.586	**0.817**	0.231	0.585	**0.811**	0.226
freetype	0.639	**0.684**	0.045	0.638	**0.658**	0.020
bloaty	0.524	**0.616**	0.092	0.522	**0.593**	0.071
php	0.576	**0.613**	0.037	0.575	**0.595**	0.020
libxml	0.472	**0.519**	0.047	0.478	**0.494**	0.016
sqlite	0.350	**0.380**	0.030	0.349	**0.350**	0.001

To test the significance of the interaction term γ_{ij}, we use the two-way ANOVA [12], which tests the null hypothesis that $\gamma_{ij} = 0$ for all $i, j \in \mathcal{M}$. ANOVA has the following three assumptions [34]: First, that the dataset observations are independent. Since AFL++ saves interesting inputs to be further mutated in the future, possibly from a different pair of mutators, the independence assumption could be violated. To test the independence assumption, we employ the *Residuals vs Fitted* plot [23]. In nine target programs, the residuals follow random patterns, indicating that the independence assumption is met [34], while in four target programs the residuals exhibit a systematic trend, indicating that the independence assumption is violated. We do not fit a model in these four programs. Second, ANOVA assumes that the residuals are normally distributed, which we check by plotting the residuals and observing a bell-shaped distribution in all nine programs [34]. Finally, ANOVA assumes homoscedasticity, which we also check using the *residuals-vs-fitted* plot [34]. We find heteroscedasticity in 3/9 programs, so we use the robust `hc3` error [38] to mitigate it.

Complementary to the ANOVA results, we calculate the goodness-of-fit, measured with R^2 and R^2_{adj} [9], of the linear model with the interaction term ($\gamma_{ij} \neq 0$) and a linear model without the interaction term ($\gamma_{ij} = 0$). A higher goodness-of-fit for the variation with the interaction term ($\gamma_{ij} \neq 0$) would indicate the interaction term affects the number of interesting inputs.

Our study focuses on the interactions between mutator types, but it is possible that interactions also exist with other factors, such as mutator locations (i.e., which part of the input is mutated). There

is no reason to believe that the mutation locations or other factors are not uniformly distributed across mutator pairs. Combined with the fact that the number of mutations in our experiments is in the order of millions, we can reasonably expect that the effect of extraneous variables such as location evens out across all mutator pairs. More formally, in our experiment, the independent variables (i.e., the factors being manipulated) are the mutator types, the dependent variable (i.e., the factor being measured) is the number of interesting inputs, and extraneous variables are other factors that could potentially influence the dependent variable but are *not manipulated*, and as such, they pose no threat for the validity of the observed interaction effect.

4.3 Results

To test the null hypothesis that $\gamma_{ij} = 0$ for all $i, j \in \mathcal{M}$, we run a two-way ANOVA on the fitted model (one model for each target program). For all target programs we get $p < 0.0001$, hence reject the null hypothesis. This should not be misinterpreted as *all* mutator pairs having a significant interaction effect, rather that there exist *some* mutator pairs with a significant interaction effect. Although two-way ANOVA also generates p-values for each of the individual hypotheses $\gamma_{ij} = 0$ for fixed i and j, using these p-values to draw inferences is a bad practice because $|\mathcal{M}|^2 = 1024$ hypotheses are tested simultaneously, hence suffering from *multiple hypothesis testing* [3]. In Section 7 we manually investigate which mutators have strong interaction effects.

Complementary to the ANOVA analysis, we show the goodness-of-fit in Table 2. We focus on R^2_{adj} because it penalizes the number of model parameters, which is higher when $\gamma_{ij} \neq 0$, but even with this penalization we see that the model with the interaction term achieves higher R^2_{adj}. This means that the interaction term affects the number of interesting inputs, which agrees with the ANOVA null hypothesis test.

Finding 1. We measure an interaction effect between two mutators on the number of interesting inputs.

5 MUOFUZZ

The establishment of the existence of an interaction effect between two mutators does not guarantee the practical significance of this effect. To investigate its value for fuzzing, **we propose a method for generating mutator sequences where the probability of selecting the next mutator is conditioned on the previously selected one**. We implement this method into MuoFuzz and compare its performance against state-of-the-art fuzzers.

5.1 Definitions and Problem Statement

The goal of a mutation strategy is to mutate a seed s by applying a sequence of mutators. This sequence is of the form $\langle m_1, m_2, \cdots, m_l \rangle$. The mutators m_n are selected from a predefined set of mutators $\mathcal{M} = \{1, 2, ..., |\mathcal{M}|\}$. For example, AFL++ defines $|\mathcal{M}| = 32$ mutators which are shown in Table 1.

We reduce the problem of generating a mutator sequence $\langle m_1, m_2, \cdots, m_l \rangle$ to the problem of learning the conditional probability $\Pr(m_n = j \mid m_{n-1} = i)$ of selecting j as the next mutator in the sequence given that the previously selected mutator is i. If we learn such a probability, we can iteratively generate a mutator sequence of arbitrary length l with the following steps: We randomly select the first mutator $m_1 \in \mathcal{M}$. Then, *for fixed* m_1, we obtain m_2 by sampling from the distribution

$$m_2 \sim \Pr(j) = \Pr(m_2 = j \mid m_1)$$

In general, we obtain m_n by sampling from the distribution

$$m_n \sim \Pr(j) = \Pr(m_n = j \mid m_{n-1})$$

where m_{n-1} is fixed from the previous step. This sampling algorithm is also known as a Markov random walk [33] on the Markov chain, where the states are the mutators and $\Pr(m_n = j \mid m_{n-1} = i)$ are the transition probabilities from state i to state j.

5.2 Overview

Our mutation strategy consists of two phases. During the **training phase** (Figure 2a), we aim to learn the conditional probability $\Pr(m_n = j \mid m_{n-1} = i)$ of selecting j as the next mutator in the sequence given that the previously selected mutator is i. We model this probability by computing the number of interesting inputs produced by all possible mutator pairs $\langle i, j \rangle$ in the first T_{train} hours of fuzzing. Then, for fixed i, we define $\Pr(m_n = j \mid m_{n-1} = i)$ to be proportional to the number of interesting inputs produced by $\langle i, j \rangle$.

After the first T_{train} hours, our fuzzer enters the **guided mutation phase** (Figure 2b). Here, it generates mutator sequences by sampling from the learned distribution $\Pr(j) = \Pr(m_n = j \mid m_{n-1} = i)$ where the mutator selected in each step is used to select the mutator of the next step. The training time T_{train} that MuoFuzz spends in the training phase before switching to the guided mutation phase is a hyperparameter that we finetune in Section 6.

5.3 Training Phase

The goal of the training phase is to learn the conditional probability $\Pr(m_n = j \mid m_{n-1} = i)$ of selecting j as the next mutator in the sequence given that the previously selected mutator is i. *We define this probability to be proportional to the number of interesting inputs generated by the sequence* $\langle i, j \rangle$; we denote this number with $N(i, j)$. This design decision is intuitive: we give a higher probability of selecting the mutator j if j has generated more interesting inputs when applied on top of mutator i. To obtain $N(i, j)$, in the first T_{train} hours we run AFL++ with the three modifications **M1**, **M2**, and **M3** we applied in Section 4:

M1: We limit the length of the mutator sequences to $l = 2$.
M2: We sample the mutators uniformly at random.
M3: We update $N(i, j)$.

The modifications **M1**–**M3** are the only changes we apply to AFL++, while all other parameters remain at their default values. After T_{train} hours, we execute the last step of the training phase: we define the probability of selecting j as the next mutator in the sequence given that the previously selected mutator is i as

$$\Pr(j) = \Pr(m_n = j \mid m_{n-1} = i) = \frac{N(i, j)}{\sum_{j=1}^{|\mathcal{M}|} N(i, j)} \tag{2}$$

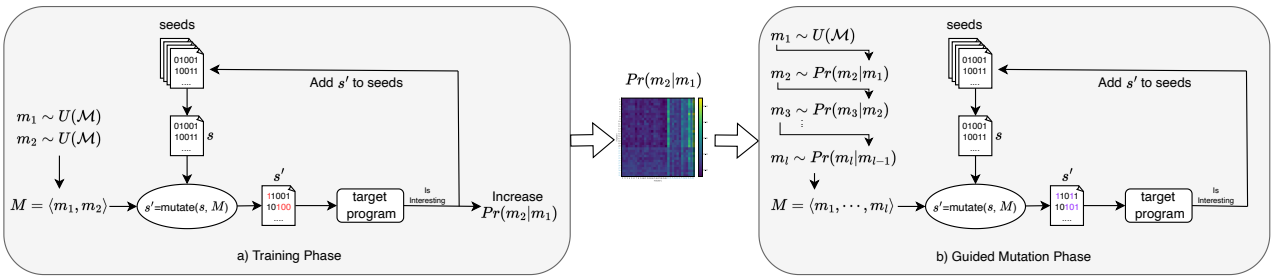

Figure 2: High-level overview of MuoFuzz.

The division with the sum turns each row into a probability distribution that sums to 1. The distribution for the `freetype` program is shown on the right side of Figure 1, while the left side shows the associated $N(i, j)$. This concludes the training phase and MuoFuzz then automatically enters the guided mutation phase. We note that our training phase is lightweight and embedded in the fuzzing loop of AFL++. This is in contrast to machine learning approaches where AFL++ is run for one hour to gather data, which are then used to train a neural network [39, 40]. Our training is as simple as increasing $N(i, j)$ whenever an interesting input is generated.

5.4 Guided Mutation Phase

Algorithm 2: Guided Mutation Phase

Input: subroutines *select_sequence_length*() and *select_first_mutator*()
Output: mutator sequence $\langle m_1, m_2, ..., m_l \rangle$

1 $l \leftarrow$ *select_sequence_length*();
2 $m_1 \leftarrow$ *select_first_mutator*();
3 $M \leftarrow \emptyset$;
4 **for** $n = 2$ **to** l **do**
5 $\quad m_n \sim \Pr(m_n \mid m_{n-1})$;
6 $\quad M \leftarrow M \cup \{m_n\}$;
7 **end**
8 **return** $M = \langle m_1, m_2, ..., m_l \rangle$;

The goal of the guided mutation phase is to generate mutator sequences $\langle m_1, m_2, ..., m_l \rangle$ by leveraging the learned probability distribution of Equation (2). We propose Algorithm 2 to generate a sequence $\langle m_1, m_2, ..., m_l \rangle$ and replace the highlighted part of Algorithm 1: First, we select the sequence length l (line 1) and the first mutator m_1 (line 2). The subroutines that perform these selections are hyperparameters and are discussed in detail below. Then, at each step $2 < n < l$ (line 4), we obtain m_n by sampling from the learned probability $\Pr(m_n \mid m_{n-1})$ (line 5), where now m_{n-1} is fixed from the previous step. In the remainder of this section, we discuss the hyperparameters of the guided mutation phase, and in Section 6 we experimentally select the best-performing values. By default, MuoFuzz uses the best-performing hyperparameters as per Section 6 unless stated otherwise.

5.4.1 Selecting the Sequence Length. Algorithm 2 leaves open the question of what the optimal length l of a mutator sequence is. We consider two options for this hyperparameter. The *first option* is to keep the AFL++ default sequence length that ranges from 2 to 16. However, due to our optimized generation algorithm, a different, potentially smaller length may be sufficient. For this reason, we propose a *second option*: treating the sequence length as a dynamically determined variable. We follow previous work [21] and model the sequence length selection as a multi-armed bandit (MAB) problem [24]. We solve this problem using the ε-greedy algorithm, a well-known heuristic solution to the MAB problem that has been applied in previous work [13]. Given a set of possible values for the sequence length $l \in L = \{2, 3, ..., 16\}$, the ε-greedy algorithm selects the best performing (i.e., the one that has generated the most interesting inputs) sequence length with probability ε, and with probability $1 - \varepsilon$ it selects the sequence length uniformly at random from L. The value of ε follows a step decay, starting from $\varepsilon = 1$ (exploration only) in the first hour of the guided mutation phase, then dropping to $\varepsilon = 0.5$ to boost exploitation. We select the ε-greedy algorithm over other heuristic MAB algorithms like Upper Confidence Bound (UCB) [2] or Thompson Sampling [24] because of its simplicity and interpretability.

5.4.2 Selecting the First Mutator. Another hyperparameter of the guided mutation phase is how to select the first mutator m_1. We investigate two options for m_1:

(a) a uniformly random selection of m_1 from $\mathcal{M}$, as is usually done in random walks [33].
(b) a weighted selection of m_1 that gives a higher probability to mutators that produced more interesting inputs.

To implement (b), for each mutator i we calculate a score

$$s_i = \sum_{j=1}^{|\mathcal{M}|} N(i, j) + \sum_{j=1}^{|\mathcal{M}|} N(j, i) - N(i, i)$$

that holds the total number of interesting inputs produced by mutator i during the training phase. The first sum corresponds to the number of interesting inputs where i was the first mutator in the sequence, the second sum corresponds to the number of interesting inputs where i was the second mutator in the sequence, and we subtract the last term because $N(i, i)$ was counted twice, one time in each sum. After computing s_i for each $i \in \mathcal{M}$, we normalize them by dividing by their sum to form a probability distribution and sample the first mutator m_1 from that distribution.

5.4.3 Using More Mutators as Context. The decision to use only the last mutator instead of the last $p > 1$ mutators is not arbitrary. Modeling the probability $Pr\left(m_n \mid m_{n-1}m_{n-2}...m_{n-p}\right)$ of selecting the mutator m_n given the previous p mutators is possible, but it suffers from the curse of dimensionality [22]: The 2-dimensional matrix $N(i, j)$ would become $p + 1$-dimensional. Even for $p = 2$,

we would end up with $32^3 = 32768$ unique mutator triplets. Due to this combinatorial explosion, most triplets remain unobserved during training because they do not produce a coverage-increasing input. This guided our design to look only at the previous mutator when selecting the next one. Nevertheless, we run experiments with $p = 2$ in Section 6.4.1 to validate our reasoning.

5.5 Alternative Designs

In our two-phase design, the probability learned in the first T_{train} hours of fuzzing guides the mutations for the remainder of the fuzzing campaign, which, as we show below, is effective in most cases. However, alternative designs that move the training phase later could be considered, and we discuss two of them here, explaining why they were not preferred. The first alternative would be to dynamically update $N(i, j)$ throughout the entire fuzzing campaign in a Multi-Armed Bandit (MAB) setting [24], as done in similar fuzzing optimizations [13, 46]. However, we showed in RQ1 that there exist interactions between mutators, which the MAB setting cannot handle because it assumes independence of mutators (arms). Another alternative would be to move or repeat the training phase later in the fuzzing campaign, because some mutator pairs could be more effective early on but less so toward the end. However, this design suffers from the limited amount of training data (i.e., interesting inputs discovered) since greybox fuzzers discover new interesting inputs near-logarithmically [4]. We experimentally validated this in initial experiments by shifting the training phase one hour later in the fuzzing campaign and observing a significant performance drop.

6 EMPIRICAL EVALUATION OF MUOFUZZ

We develop our novel mutation strategy into a fuzzer named MuoFuzz by replacing the mutation strategy of AFL++ and compare its performance to state-of-the-art baselines. Since achieved code coverage is the most widely used measure of fuzzer performance [37], we compare the code coverage of MuoFuzz against the baselines to answer our first research question.

> **RQ2**. How does MuoFuzz compare to AFL++ and MOPT in terms of code coverage?

Coverage alone is not a sufficient measure of performance [17, 20, 37]. It is only useful when accompanied by other measures, mainly the number of bugs found. Our next research question compares the number of bugs found by MuoFuzz against the baselines.

> **RQ3**. How does MuoFuzz compare to AFL++ and MOPT in terms of bugs found?

Finally, we follow recent work on fuzzer evaluation [20, 32, 37] that strongly recommends performing an ablation study when evaluating a new fuzzer for two main reasons. First, to increase the confidence that the success of MuoFuzz comes from leveraging the information about the previously selected mutator and is not a byproduct of some latent design decision. Second, to understand how different hyperparameters affect the performance. For these reasons, we design an ablation study to extensively evaluate MuoFuzz under different hyperparameters.

> **RQ4**. How do different hyperparameters affect the performance of MuoFuzz?

6.1 Experimental Setup

The empirical comparison of two fuzzers is a challenging problem, mainly due to the inherent randomness of fuzzing. Klees et al. [20] propose that the fuzzers run for 24 hours and the experiment is repeated 20 to 30 times for a single program. Then, statistical tests like the Mann-Whitney U test [30] should be performed to test the hypothesis that one fuzzer achieves higher coverage than the other, and the p-values should be reported along with the effect size.

6.1.1 Implementation. We implement MuoFuzz on top of the latest version of AFL++ at the time, which was 4.21a. Hence, the baselines are also versioned on AFL++ 4.21a. We open source our implementation to be independently assessed by the community [19].

6.1.2 Baselines. A best practice when selecting baselines is to always compare against the fuzzer on top of which the new fuzzer is built [20, 37]. In our case, we build on top of AFL++, so this is our first baseline. Additionally, it is suggested to compare against state-of-the-art fuzzers that propose different approaches to the same problem, in our case the mutation strategy. MOPT [29] proposes a mutation strategy that optimizes the selection probability of each mutator *in isolation*, in contrast to our approach which optimizes the probability of mutator pairs, which makes it a well-suited baseline. Moreover, Schloegel et al. [37] found that MOPT is consistently selected as a baseline lately, so we select MOPT as a second baseline. We use the AFL++ version of MOPT, since the original implementation of MOPT is based on AFL, which would put MOPT at a disadvantage. We intended to use SeamFuzz [25] as an additional baseline for the same reason as MOPT, but their replication package is based on an old version of AFL++ v3.15, which would also put SeamFuzz at a great disadvantage. After our inquiry to the authors, they assured us that they are working on porting SeamFuzz to the latest version in the near future.

6.1.3 Benchmarks. The most widely used fuzzing benchmark is FuzzBench [31, 37], which contains a diverse set of real-world open-source C programs representing a variety of application domains. FuzzBench automates the execution and coverage measurement, ensuring reproducible and consistent results. For these reasons, we use FuzzBench in our experiments. However, FuzzBench does not report the number of bugs found. Hence, we also use MAGMA [16], a benchmark of real-world bugs that MAGMA developers manually ported to the latest version of each program.

6.1.4 Infrastructure. All experiments take place in 16 identical virtual machines (VMs) running Ubuntu 22.04, each having an AMD EPYC 7702 processor with 32 CPUs, 125 GB RAM, and a 100 GB disk. We restrict experiments of the same target program to the same set of VMs, for example, we use only VM1 and VM2

Table 3: Median coverage and st.d. after 24 hours of MuoFuzz (ours), AFL++, and MOPT on FuzzBench.

Target	AFL++	MOPT	MuoFuzz	p_{AFL++}	$\hat{A}_{12}$	p_{MOPT}	$\hat{A}_{12}$
bloaty	5,990 ± 78	5,216 ± 332	**6,028** ± 73	0.0185**	0.65	0.0000***	0.99
curl	10,867 ± 102	10,773 ± 94	**10,912** ± 82	0.0126**	0.67	0.0000***	0.89
freetype	11,006 ± 484	**11,682** ± 327	11,491 ± 397	0.0006***	0.73	0.9987	0.31
json	**520** ± 0	**520** ± 1	**520** ± 0	NaN	NaN	0.0926*	0.56
lcms	1,956 ± 204	2,002 ± 218	**2,004** ± 348	0.3482	0.52	0.9525	0.40
libpcap	2,802 ± 125	2,780 ± 113	**2,840** ± 131	0.0323**	0.61	0.0077**	0.64
libpng	2,004 ± 3	2,005 ± 26	**2,006** ± 3	0.0781*	0.59	0.2547	0.53
libxml	19,212 ± 244	19,247 ± 891	**19,256** ± 215	0.7313	0.46	0.2214	0.55
openssl	5,818 ± 7	**5,828** ± 6	5,827 ± 5	0.0000***	0.82	0.8350	0.42
php	**16,712** ± 79	16,653 ± 23	16,689 ± 90	0.4923	0.50	0.0147**	0.69
proj4	6,872 ± 134	6,981 ± 179	**7,068** ± 175	0.0000***	0.78	0.0559*	0.59
re2	2,876 ± 4	2,874 ± 5	**2,879** ± 5	0.0022***	0.68	0.0002***	0.75
sqlite	19,907 ± 801	18,954 ± 798	**20,091** ± 254	0.0003***	0.76	0.0000***	0.97

Highest median = **bold**, second highest = underlined. ***: $p < 0.01$, **: $p < 0.05$, *: $p < 0.1$.

Table 4: Time needed (hours) for MuoFuzz to reach the coverage achieved by the baselines after 24 hours.

	Time to final AFL++ cov.			Time to final MOPT cov.		
Target	By AFL++	By MuoFuzz	δ	By MOPT	MuoFuzz	δ
bloaty	23.8	20.0	**3.8**	23.5	1.5	**22.0**
curl	22.5	15.8	**6.7**	23.8	7.0	**16.8**
freetype	23.8	12.0	**11.8**	17.8	23.8	-6.0
lcms	23.5	13.8	**9.7**	23.0	21.5	**1.5**
libpcap	23.8	19.2	**4.6**	23.8	16.5	**7.3**
libpng	17.2	7.8	**9.4**	19.2	11.0	**8.2**
libxml	23.8	22.8	**1.0**	23.8	23.2	**0.6**
openssl	15.5	9.0	**6.5**	18.2	19.5	-1.3
php	22.0	23.8	-1.8	23.8	19.8	**4.0**
proj4	23.8	14.2	**9.6**	23.8	18.5	**5.3**
re2	23.8	13.5	**10.3**	21.5	9.2	**12.3**
sqlite	23.8	19.0	**4.8**	22.0	8.8	**13.2**
Median	23.8	15.0	6.6	23.3	16.5	6.3

for the experiments on `proj4` to further ensure fairness (although the VMs are identical). Only 24 out of 32 cores (~80%) are used simultaneously.

6.2 RQ2 — Code Coverage

FuzzBench contains 28 programs but is continuously maintained so some programs that existed in older versions may not exist anymore. Evaluating in all 28 programs is computationally intensive: as reported by Schloegel et al. [37], fuzzers in top venues of the last five years are evaluated on 8.9 programs on average. To avoid bias in the program selection, we use the same programs as two recent fuzzing papers that also used FuzzBench [25, 35]. Five of the programs used by Li et al. [25] are available in the latest FuzzBench version, as well as eight of the programs used by Qian et al. [35]. We use their union, totaling thirteen programs.

We run each fuzzer on each program for 24 hours and 30 trials while keeping the default initial seeds of FuzzBench. We report the median coverage and the standard deviation after 24 hours in Table 3. To compare MuoFuzz with a baseline, we employ the Mann-Whitney U test with the null hypothesis that MuoFuzz does not achieve higher coverage than the baseline. Since we have two baselines, AFL++ and MOPT, we run the test two times, one against AFL++ and one against MOPT. The resulting p-values, p_{AFL++} and p_{MOPT} respectively, are also shown in Table 3, along with the associated Vargha-Delaney effect size [43] ($\hat{A}_{12}$), which denotes the probability that a random trial of MuoFuzz will achieve higher coverage than a random trial of the baseline fuzzer. We see that MuoFuzz achieves higher coverage than AFL++ in 9/13 programs and than MOPT in 8/13 programs with statistical significance.

Table 3 answers the question of how many more branches can MuoFuzz discover in a 24-hour campaign, but does not tell us anything about how faster it reaches that coverage compared to the baselines. This is particularly important as Böhme et al.[4] found that the cost of discovering new branches in greybox fuzzers increases *exponentially* with time. In other words, even a seemingly small increase in the final coverage after 24 hours may be hard to achieve. Hence, they suggest a complementary metric when comparing coverage between two fuzzers: How long does it take one fuzzer to reach the final coverage the other fuzzer achieves after 24 hours? For example, after 24 hours in the *curl* program, AFL++ achieves a median coverage of 10 867 while MuoFuzz achieves 10 912, which may seem like a not-so-important increase at first glance (0.9%). However, MuoFuzz reaches 10 867 coverage after only 15.8 hours (median over 30 trials), while AFL++ first reaches that number after 22.5 hours (its median coverage does not increase in the last 1.5 hours). We provide this metric for all target programs in Table 4. We find that MuoFuzz reaches the 24-hour-coverage of AFL++ after 15 hours (median across all programs) and the 24-hour-coverage of MOPT after 16.5 hours.

To better understand when MuoFuzz yields coverage improvements, we analyzed the programs in which MuoFuzz did not achieve a statistically significant improvement in coverage over either of the baselines. For the `lcms`, `libpng`, and `openssl` programs, the interaction effect could not be observed already from our empirical analysis in RQ1: The data for these three programs violated the independence assumption and thus we did not run the ANOVA analysis. Moreover, for `libpng` and `openssl`, the learned probability matrix does not show clear underlying patterns, in contrast to the other programs where such patterns are visible, indicating the absence of an interaction effect. For `freetype`, `libxml`, and `php`, the goodness-of-fit (R^2_{adj}) in Table 2 is 0.658, 0.595, and 0.494 respectively, indicating that the linear model of Equation (1) only partially captures $N(i, j)$ and, as a consequence, the interaction effect may not be strong. In all programs where $R^2_{adj} > 0.8$, MuoFuzz outperforms both baselines with statistical significance, except for AFL++ in `json`. For `json`, all fuzzers reach the maximum coverage of 520 after only 15 minutes, except for some runs of MOPT. This early saturation suggests that the mutation strategy does not affect the coverage in the `json` program and that other techniques, such as symbolic execution, may be required to explore `json` in more depth.

Finding 2. MuoFuzz achieves higher coverage than AFL++ and MOPT in 9/13 and 8/13 programs respectively, while for the remaining programs we observed a weak interaction effect that did not lead to improved performance. MuoFuzz needs 15 and 16.5 hours on average to reach the 24-hour coverage of AFL++ and MOPT respectively.

6.3 RQ3 — Bugs Found

In the second set of experiments we compare the bug-finding ability of MuoFuzz to the baseline fuzzers. The latest version of MAGMA (v.1.2) comes with eight target programs, so we use all of them. A target program may have more than one driver file in MAGMA, in which case we fuzz all of them. The list of programs along with their driver files is available in the MAGMA repository.

We run each fuzzer on each benchmark for 24 hours and 20 trials, while keeping the default initial seeds of MAGMA. MuoFuzz detects four bugs that AFL++ cannot detect (`XML002`, `SSL009`, `LUA002`, and `SQL010`) while AFL++ detects only one bug that MuoFuzz cannot detect (`TIF001`). In other words, MuoFuzz finds three more bugs than AFL++. When we look at `TIF001`, we see that neither MuoFuzz nor MOPT found the bug. Only 1 of the 20 trials of the baseline AFL++ found it, quite possibly due to the inherent randomness of fuzzing. In contrast, MuoFuzz found the three bugs that AFL++ did not find in 3/20, 3/20, and 2/20 trials respectively. Regarding MOPT, it finds the same number of bugs as MuoFuzz, but `SSL001` is only found by MOPT and `SQL010` is only found by MuoFuzz. This suggests that although the total number of bugs is the same, MuoFuzz may trigger a different, unique behaviour compared to MOPT. The bug `SSL001` that MuoFuzz missed belongs to the *openssl* program, where the interaction effect is weak and no clear pattern is evident, as shown in Section 6.2.

We also compare the number of unique bugs triggered by each fuzzer averaged across the 20 trials. In the `libsndfile` program, all three fuzzers trigger exactly seven bugs in all trials. In the remaining seven programs, MuoFuzz ranks first in four, MOPT ranks first in two, and AFL++ ranks first in one. However, the differences in the average number of unique bugs are not statistically significant according to the Mann-Whitney U test [30]. We see a similar pattern in the time-to-bug (considering the bugs that all three fuzzers found), where no single fuzzer consistently outperforms the other two. The time-to-bug as well as the number of average unique bugs for every fuzzer and program are provided in our replication package for space reasons [19].

> **Finding 3**. MuoFuzz detects three more bugs than AFL++. It also detects the same number of bugs as MOPT, but each fuzzer discovered a distinct bug, with MuoFuzz missing a bug in a program where the interaction effect was weak. The above suggest that leveraging the interaction effect, when it exists, can trigger unique program behaviour.

To better understand how the design choices and hyperparameters of MuoFuzz affect its performance across different target programs, we conduct a detailed ablation study in Section 6.4.

6.4 RQ4 — Ablation Study

The goal of the ablation study is to better understand the effect of various components and hyperparameters of the proposed fuzzer. Since fuzzers are complex systems comprised of many components, some components may contribute most to the performance while others do not contribute at all. In the rest of this section, we run experiments to understand how different hyperparameters of MuoFuzz contribute to its performance. We focus on FuzzBench because it yielded more decisive results in the first set of experiments. We report the results in Table 5.

6.4.1 Using More Mutators as Context. MuoFuzz takes into account only the last mutator when predicting the next. We reasoned about the disadvantages of using $p > 1$ previous mutators in Section 5.4.3; we experimentally validate this reasoning by implementing a variation of MuoFuzz that takes into account $p = 2$ previous mutators to select the next one. We modify the logic as described in Section 5.4.3 and call this variation $\text{MuoFuzz}_{p=2}$. Table 5 shows that $\text{MuoFuzz}_{p=2}$ underperforms MuoFuzz in all target programs. By manually analyzing $N(i, j, k)$ for each program, we see that the sparsity [8] ranges from 0.003 to 0.1, which validates that the learning suffers from the curse of dimensionality [22].

6.4.2 Selecting the Length of the Mutator Sequence. The length l of the mutator sequence $\langle x_1, x_2, ..., x_l \rangle$ is another hyperparameter of MuoFuzz. In Section 5 we described the MAB algorithm MuoFuzz uses to determine the length l, and here we investigate its impact on MuoFuzz performance. To do so, we run a variation of MuoFuzz where l follows the default AFL++ distribution (see Section 2). We call this variation $\text{MuoFuzz}_{\text{default length}}$.

From Table 5, we see that $\text{MuoFuzz}_{\text{default length}}$ performs better than or equal to MuoFuzz in four out of thirteen programs. For the other nine programs, MuoFuzz performs better. Hence, we conclude that the MAB-optimized sequence length contributes to the performance of MuoFuzz.

6.4.3 Selecting the First Mutator. Our proposed algorithm for generating mutator sequences specifies how to select the next mutator x_n given the previous mutator in the sequence x_{n-1}. This leaves open the question of how to select the first mutator x_1 of the sequence $\langle x_1, x_2, ..., x_l \rangle$. We investigate two viable answers:

- a uniformly random selection of m_1 (i.e., the default variation),
- a weighted selection of m_1 that gives a higher probability to mutators that have produced more interesting inputs.

We call the latter $\text{MuoFuzz}_{\text{weighted } m_1}$. We see that MuoFuzz performs better than or equal to $\text{MuoFuzz}_{\text{weighted } m_1}$ in ten out of thirteen programs. Hence, we conclude that a uniformly random selection of m_1 is preferred over a weighted selection. This may seem counterintuitive since the weighted selection starts the sequence with a more "promising" mutator. Our interpretation is that the weighted selection makes the first mutator more deterministic, reducing the overall randomness of the mutator sequence, which in turn reduces the fuzzer exploration.

6.4.4 Using a Random Matrix. To create a meaningful baseline, we replace the matrix $N(i, j)$ that holds the number of interesting inputs generated by combining mutator i with mutator j with a random matrix. We call this resulting baseline $\text{MuoFuzz}_{\text{random}}$ and show its performance in Table 5. We see that $\text{MuoFuzz}_{\text{random}}$ performs worse than MuoFuzz in all target programs, which increases our confidence that the performance of MuoFuzz is a result of its design and not of randomness.

6.4.5 Selecting the Training Time. Previous work on machine learning based fuzzers [39, 40] also follows a two-phase process like ours,

Table 5: Median coverage and standard deviation after 24 hours for MuoFuzz variations.

Target	MuoFuzz Variations																							
	original			p=2			default length			weighted m_1			random			t=2			t=0.5			cross program		
proj4	7,068	±	175	5,490	±	399	6,993	±	147	6,982	±	172	4,991	±	346	6,974	±	153	7,039	±	181	6,991	±	145
curl	10,912	±	82	10,407	±	85	10,926	±	89	10,876	±	94	10,322	±	99	10,860	±	94	10,876	±	54	10,858	±	103
freetype	11,491	±	397	10,960	±	508	11,240	±	429	11,204	±	419	10,544	±	427	11,213	±	366	11,366	±	290	11,520	±	302
bloaty	6,028	±	73	5,408	±	276	6,341	±	79	5,971	±	93	5,244	±	158	6,018	±	106	6,015	±	102	5,960	±	104
php	16,689	±	90	16,054	±	75	16,676	±	68	16,651	±	76	16,054	±	75	16,658	±	59	16,593	±	72	16,536	±	51
libxml	19,256	±	215	14,783	±	481	19,166	±	268	19,246	±	225	18,994	±	1,800	19,213	±	278	19,236	±	177	19,054	±	250
sqlite	20,091	±	254	17,028	±	729	19,934	±	318	20,196	±	691	16,675	±	1,324	20,188	±	851	20,176	±	397	20,100	±	968
libpng	2,006	±	3	1,980	±	19	2,005	±	3	2,004	±	3	1,977	±	16	2,006	±	18	2,006	±	24	2,005	±	24
libpcap	2,840	±	131	2,417	±	122	2,830	±	106	2,852	±	132	2,307	±	114	2,841	±	144	2,817	±	135	2,769	±	105
openssl	5,827	±	5	5,821	±	7	5,822	±	6	5,824	±	7	5,822	±	6	5,824	±	6	5,804	±	16	5,819	±	8
lcms	2,004	±	348	1,964	±	211	1,958	±	218	2,010	±	239	1,585	±	369	2,004	±	223	1,988	±	175	2,002	±	170
re2	2,879	±	5	2,836	±	18	2,877	±	4	2,878	±	5	2,860	±	29	2,880	±	3	2,877	±	4	2,878	±	4
json	520	±	0	520	±	0	520	±	0	520	±	0	520	±	0.2	520	±	0	520	±	0	520	±	0

Variations that perform better than the original are shown underlined.

where the first hours of the fuzzing budget are dedicated for training and the rest of them for inference. For example, Neuzz [40] runs AFL for one hour to collect data about which bytes of the seed are more likely to yield an interesting input when mutated. For this reason, we select $T_{train} = 1$ hour as a starting option for our training phase. To understand the impact of T_{train}, we also run two variations with $T_{train} = 0.5$ hours and $T_{train} = 2$ hours, named MuoFuzz$_{t=0.5}$ and MuoFuzz$_{t=2}$ respectively. We see that MuoFuzz (with $T_{train} = 1$) performs at least as good as MuoFuzz$_{t=2}$ in ten out of thirteen target programs and at least as good as MuoFuzz$_{t=0.5}$ in eleven out of thirteen programs.

6.4.6 Cross-Program Generalization of Learned Probability. One question that arises is whether the learned probabilities in one target program are transferable to another program. In other words, whether there is a universal probability of mutation pairs that works well in all programs. To answer this, we first qualitatively analyze the learned probability of the thirteen programs. We find that in *nine* programs the learned probability follows a similar pattern as the one shown in Figure 1. For the other *four* programs (`libpng`, `libpcap`, `openssl`, and `lcms`), we see four distinct patterns. We randomly select one of the nine programs (`freetype`) and use the probability learned on that program to guide the mutations in the other programs. The training phase is disabled since no learning is required and the guided mutation phase takes up the whole fuzzing campaign (24 hours); We call this variation MuoFuzz$_{\text{cross program}}$.

From Table 5, we see that in the four programs that exhibit a different pattern than `freetype`, the performance of MuoFuzz$_{\text{cp}}$ is lower than or equal to MuoFuzz$_{\text{random}}$. In the other nine programs, MuoFuzz$_{\text{cp}}$ performs worse than MuoFuzz$_{\text{random}}$ only in two programs, while it performs even better than MuoFuzz in two other programs. These results indicate that, although cross-program generalization is possible between some (not all) programs, training in each program individually generally leads to better results.

We notice that in `freetype`, MuoFuzz$_{\text{cp}}$ performs better than MuoFuzz. This makes sense: the training phase, where only two mutators are stacked together, is disabled and the fuzzer remains in the guided mutation phase for 24 hours instead of 23. This suggests that the training phase slows down MuoFuzz, making the effectiveness of the guided mutation phase even higher. To validate, we check the performance of MuoFuzz after one hour (the end of the training phase) for the ten programs in which MuoFuzz outperformed the baselines in Section 6.2. We find that only in four of them MuoFuzz was better after one hour; in the other six programs, MuoFuzz started from a disadvantage and surpassed the baselines in the next 23 hours of the guided mutation phase.

The ablation study increases our confidence that MuoFuzz performance is a result of the effective harness of the interaction effect between mutators and quantifies the effect of hyperparameters.

7 DISCUSSION

In this section, we interpret the outcomes of the training phase of MuoFuzz (Section 7.1) and discuss the implications of our findings to future research (Section 7.2).

7.1 What does MuoFuzz learn during training?

The performance of MuoFuzz motivates a qualitative analysis of the learned probability $\Pr(m_n \mid m_{n-1})$ to see which mutators (or mutator families) are better combined with which. For example, the learned probability for the `freetype` target program is shown in Figure 1 (right). *Unit* mutators that apply simple, lightweight transformations have a black label font, while *chunk* mutators, that disruptively transform the seed have a blue label font. We observe the following pattern: if the first mutator (row) is a unit mutator, the second mutator (column) is more likely to be a chunk mutator. On the other hand, if the first mutator is a chunk mutator, the second mutator follows a roughly uniform distribution. A similar pattern is followed in eight other target programs. For the target program `openssl`, however, we observe a different pattern: unit mutators dominate over chunk mutators. This means that a mutator sequence sampled from this learned probability will have mostly unit mutators. This could be because the input has a strict format that easily breaks with chunk mutators. A similar, but weaker pattern is observed for the `libpng` target program. Finally, for the `libpcap` target program the pattern has nothing to do with unit or chunk mutators. Specifically, mutators that apply simple arithmetics (IDs 7-18) tend to work better when stacked on top of each other, and also when followed by clone mutators (IDs 29 and 30).

We note that the space of all possible mutator sequences is the same (or a subset, if $N(i, j)$ contains zeros) for MuoFuzz and the baselines. This means that MuoFuzz does not generate a new sequence that cannot be generated by the baselines, rather it generates the more interesting sequences earlier in the fuzzing campaign.

7.2 Implications for Future Research

We mention here the similarity of generating mutator sequences with our proposed method to generating text (token sequences, where tokens can be characters, words, or any other token type) using bigram language models (LMs) [7]. The goal of a LM is to learn the conditional probability $P(x_n \mid x_1 x_2 ... x_{n-1})$ of the token x_n given the previously generated tokens $x_1 x_2 ... x_{n-1}$. Bigram language models, an early predecessor of modern powerful LMs [1, 42], approximate this probability by looking only at the last token x_{n-1}. The training process consists of computing the frequency of all bigrams $\langle i, j \rangle$ appearing in the training set. The inference of the next character consists of feeding the previously generated character to the probability distribution defined by normalizing the calculated bigram frequencies. Although the problem domain is different, our two-phase mutation strategy is inspired—up to a certain degree—by the bigram language models. This similarity yields the question of whether we can apply more advanced text generation techniques to the problem of generating mutator sequences. For example, to train a Recurrent Neural Network (RNN) [10] or a Transformer [44] on a dataset of successful mutator sequences. Information about the seed on which the successful mutator was applied could also be encoded as additional context. These ideas are applicable in light of our results, which show that information about the previous mutator can help select the next mutator.

Another line of future work stems from our finding that selecting the first mutator at random tends to outperform a weighted selection, which we interpret as increased randomness in the generated sequences. The overall randomness of our mutation strategy can be increased by using a different normalization than simply dividing by the sum in Equation (2). For example, the softmax [14] normalization comes with the temperature hyperparameter T, where higher T yields less randomness, and lower T yields higher randomness. Controlling this hyperparameter would be a way to account for the exploration/exploitation trade-off in the mutation strategy.

Finally, the idea of considering the previous state in a mutator sequence can be generalized beyond mutators: it would be interesting to predict the next position to mutate, given the position mutated by the previous mutator in the sequence. This work would complement related work that deals with the problem of targeting mutators to specific positions of the seed [26, 36, 40].

8 THREATS TO VALIDITY

We present here the threats to the validity of our study.

External Validity. The target programs we use in our experiments are open-source C libraries, hence we do not make claims beyond that. Overfitting a specific set of target programs is a threat to validity in fuzzer evaluation [37]. To mitigate this threat, we systematically reason about the selection of our target programs. MAGMA comes with nine target programs, which is computationally affordable, so we use all of them. FuzzBench, on the other hand, comes with 28 target programs, which is beyond our computational budget. For this reason, we randomly select 13 target programs, by considering the union of the target programs used in two recent related works. Not comparing to meaningful state-of-the-art fuzzers is another threat, which we mitigate by comparing a) against AFL++, which is the fuzzer we build on top of, and b) against MOPT, which optimizes the selection probability of each mutator in isolation.

Internal Validity. The correctness of the implementation of any new tool is a threat to validity, which we mitigate by open-sourcing our code to be assessed by the community [19]. We run statistical tests to increase the confidence that the observed performance difference is not a result of the inherent randomness of fuzzing but of the more effective mutator sequences. Also, we run an ablation study to increase the confidence that the performance of MuoFuzz comes from leveraging the interaction effect between mutators. Finally, measuring fuzzer performance based on proxy measures such as code coverage alone poses a threat to fuzzer evaluation [37]. We mitigate this threat by running experiments on bugs from real-world programs using the MAGMA benchmark.

9 CONCLUSION

This work investigates and proposes a method to leverage the interaction effect between mutators in greybox fuzzers.

We investigate the interaction effect by fitting a linear model to a dataset that contains the effectiveness for all possible mutator pairs of AFL-based fuzzers. We find that the interaction term of the model explains a statistically significant portion of the model's variance, thus it can affect the effectiveness of mutator sequences.

We investigate whether this finding can be leveraged in practice by developing MuoFuzz, a fuzzer that generates mutator sequences by using information about the previously selected mutator to select the next one. Our empirical evaluation shows that MuoFuzz outperforms AFL++ and MOPT in terms of achieved code coverage, and also detects a bug that none of these fuzzers was able to detect. Given that AFL++ uses a fixed selection probability and MOPT optimizes the selection probability of each mutator in isolation, our results show that mutator strategies that account for the interaction effect can be more effective.

ACKNOWLEDGMENTS

K. Kitsios and A. Bacchelli gratefully acknowledge the support of the Swiss National Science Foundation through the SNSF Project 200021_197227. The authors would also like to thank the Swiss Group for Original and Outside-the-box Software Engineering (CHOOSE) for sponsoring the trip to the conference.

REFERENCES

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report.

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

[3] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society: Series B (Methodological)* 57, 1 (1995), 289–300. https://doi.org/10.1111/j.2517-6161.1995.tb02031.x

[4] Marcel Böhme and Brandon Falk. 2020. Fuzzing: On the exponential cost of vulnerability discovery. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 713–724.

[5] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2329–2344.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1032–1043.

[7] Peter F. Brown, Vincent J. Della Pietra, Peter V. deSouza, Jennifer C. Lai, and Robert L. Mercer. 1992. Class-Based n-gram Models of Natural Language. *Computational Linguistics* 18, 4 (1992), 467–479. https://www.aclweb.org/anthology/J92-4003/

[8] Timothy A Davis. 2006. *Direct methods for sparse linear systems*. SIAM.

[9] Norman R. Draper and Harry Smith. 1998. Applied Regression Analysis. *Wiley Series in Probability and Statistics* 3 (1998).

[10] Jeffrey L. Elman. 1990. Finding Structure in Time. *Cognitive Science* 14, 2 (1990), 179–211. https://doi.org/10.1207/s15516709cog1402_1

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

[12] R. A. Fisher. 1925. Statistical Methods for Research Workers. *Edinburgh Oliver and Boyd* 12 (1925), 356–369.

[13] Vasudev Gohil, Rahul Kande, Chen Chen, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. 2024. Mabfuzz: Multi-armed bandit algorithms for fuzzing processors. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org

[15] Google. n.d.. libFuzzer – A Library for Coverage-Guided Fuzz Testing. https://llvm.org/docs/LibFuzzer.html. Accessed: 2024-08-25.

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

[17] Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th international conference on software engineering*. 435–445.

[18] Patrick Jauernig, Domagoj Jakobovic, Stjepan Picek, Emmanuel Stapf, and Ahmad-Reza Sadeghi. 2022. DARWIN: Survival of the fittest fuzzing mutators. *arXiv preprint arXiv:2210.11783* (2022).

[19] Konstantinos Kitsios, Marcel Böhme, and Alberto Bacchelli. 2025. *Replication Package for "On Interaction Effects in Greybox Fuzzing"*. https://doi.org/10.5281/zenodo.17391101

[20] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2123–2138.

[21] Yuki Koike, Hiroyuki Katsura, Hiromu Yakura, and Yuma Kurogome. 2022. Slopt: Bandit optimization framework for mutation-based fuzzing. In *Proceedings of the 38th Annual Computer Security Applications Conference*. 519–533.

[22] Mario Köppen. 2000. The curse of dimensionality. In *5th online world conference on soft computing in industrial applications (WSC5)*, Vol. 1. 4–8.

[23] Michael H. Kutner, Christopher J. Nachtsheim, John Neter, and William Li. 2005. *Applied Linear Statistical Models* (5th ed.). McGraw-Hill/Irwin, Boston, MA.

[24] Tor Lattimore and Csaba Szepesvári. 2020. *Bandit Algorithms*. Cambridge University Press, Cambridge, UK. https://tor-lattimore.com/downloads/book/book.pdf

[25] Myungho Lee, Sooyoung Cha, and Hakjoo Oh. 2023. Learning seed-adaptive mutation strategies for greybox fuzzing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 384–396.

[26] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 475–485.

[27] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. 2019. 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*. 533–544.

[28] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Jianzhong Liu, and Yu Jiang. 2024. Dodrio: Parallelizing Taint Analysis Based Fuzzing via Redundancy-Free Scheduling. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 244–254.

[29] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1949–1966. https://www.usenix.org/conference/usenixsecurity19/presentation/lyu

[30] Henry B. Mann and Donald R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50–60. https://doi.org/10.1214/aoms/1177730491

[31] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. 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*. 1393–1403.

[32] Maria-Irina Nicolae, Max Eisele, and Andreas Zeller. 2023. Revisiting neural program smoothing for fuzzing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 133–145.

[33] J. R. Norris. 1998. *Markov Chains*. Cambridge University Press, Cambridge, UK.

[34] Pennsylvania State University. 2025. ANOVA Assumptions. https://online.stat.psu.edu/stat500/lesson/10/10.2/10.2.1 Accessed: 2025-03-05.

[35] Ruixiang Qian, Quanjun Zhang, Chunrong Fang, Ding Yang, Shun Li, Binyu Li, and Zhenyu Chen. 2024. DiPri: Distance-based Seed Prioritization for Greybox Fuzzing. *ACM Transactions on Software Engineering and Methodology* (2024).

[36] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware evolutionary fuzzing.. In *NDSS*, Vol. 17. 1–14.

[37] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. 2024. SoK: Prudent Evaluation Practices for Fuzzing. *arXiv preprint arXiv:2405.10220* (2024).

[38] Skipper Seabold and Josef Perktold. 2010. statsmodels: Econometric and statistical modeling with python. In *9th Python in Science Conference*. https://www.statsmodels.org/stable/generated/statsmodels.stats.anova.anova_lm.html

[39] Dongdong She, Rahul Krishna, Lu Yan, Suman Jana, and Baishakhi Ray. 2020. 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*. 737–749.

[40] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. Neuzz: Efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 803–817.

[41] Robert Swiecki. 2015. Honggfuzz: A general-purpose, easy-to-use fuzzer with simple, command-line interface. https://github.com/google/honggfuzz. Accessed: 2024-08-25.

[42] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. LLaMA: Open and Efficient Foundation Language Models. *arXiv preprint arXiv:2302.13971* (2023).

[43] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NeurIPS)*. Curran Associates, Inc., 6000–6010. https://arxiv.org/abs/1706.03762

[45] Xiaohong Wang, Chengcheng Hu, Ruopeng Ma, et al. 2021. CMFuzz: context-aware adaptive mutation for fuzzers. *Empirical Software Engineering* 26, 10 (2021), 1–28. https://doi.org/10.1007/s10664-020-09927-3

[46] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. 2022. One fuzzing strategy to rule them all. In *Proceedings of the 44th International Conference on Software Engineering*. 1634–1645.

[47] Michal Zalewski. 2014. American Fuzzy Lop (AFL). https://github.com/google/AFL. Accessed: 2024-08-25.

[48] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Christian Holler, Caroline Lemieux, and Zhendong Su. 2024. Greybox Fuzzing. In *The Fuzzing Book*, Andreas Zeller, Rahul Gopinath, Marcel Böhme, Christian Holler, Caroline Lemieux, and Zhendong Su (Eds.). Fuzzingbook.org. https://www.fuzzingbook.org/html/GreyboxFuzzer.html

[49] Han Zheng, Flavio Toffalini, Marcel Böhme, and Mathias Payer. 2025. MendelFuzz: The Return of the Deterministic Stage. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 44–64.