Empirical Software Engineering (2021) 26: 10
https://doi.org/10.1007/s10664-020-09927-3

CMFuzz: context-aware adaptive mutation for fuzzers

Xiajing Wang[1] · Changzhen Hu[1] · Rui Ma[1] · Donghai Tian[1] · Jinyuan He[1]

Accepted: 17 December 2020 / Published online: 16 January 2021
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC part of Springer Nature 2021

Abstract

Mutation-based fuzzing is a simple yet effective technique to discover bugs and security vulnerabilities in software. Given a set of well-formed initial seeds, mutation-based fuzzers continually generate interesting seeds by applying specific mutation strategy in order to maximize code coverage or the number of unique bugs explored at any point-in-time. However, existing fuzzers remain limited in the paths it could cover since it simply follows a uniform distribution to choose mutation operators. In this paper, we proposed a novel context-aware adaptive mutation scheme, namely CMFuzz, which utilizes a contextual bandit algorithm LinUCB to effectively choose optimal mutation operators for various seed files. To this end, CMFuzz dynamically extracts and encodes file characteristics, which allows mutation-based fuzzers to perform context-aware mutation. We apply this scheme on top of several state-of-the-art fuzzers, i.e., PTfuzz, AFL, and AFLFast, and implement CMFuzz-PT, CMFuzz-AFL, and CMFuzz-AFLFast, respectively. We conduct evaluation on 12 real-world open source applications and LAVA-M dataset against their counterparts. Extensive evaluations demonstrate that CMFuzz-based fuzzers achieve higher code coverage and find more crashes at a faster rate than their counterparts on most cases. Furthermore, we also utilize other mainstream bandit algorithms, e.g., Thompson Sample and epsilon-greedy, and implement Thompson-PT and Greedy-PT based on PTfuzz to examine the performance of proposed model. CMFuzz-PT significantly outperforms Thompson-PT especially in terms of unique crashes and paths, i.e., found 1.79× unique crashes and 1.29× unique paths on average. Compared to Greedy-PT, our approach still increases the amount of unique crashes and paths by 1.11× and 1.05×, respectively.

Keywords Fuzzing · Seed mutation · Context-aware · Bandit algorithm

Communicated by: Dan Hao

✉ Rui Ma
mary@bit.edu.cn

Extended author information available on the last page of the article

1 Introduction

Mutation-based fuzzing is a state-of-the-art program testing methodology in finding security bugs and vulnerabilities, widely adopted by the academic research (Woo et al. 2013; Cha et al. 2015; Han and Cha 2017; Li et al. 2017) as well as industry such as the mainstream community Adobe (Adobe 2019) and Google (Google 2019a, 2019b). Most notably, one particular fuzzer, AFL (Zalewski 2019), has had a significant impact on the security landscape thanks to its ease-of-use. In general, it utilizes a set of legitimate samples as seed inputs and continuously mutates them to probe various hard-to-reach program paths and trigger potential vulnerabilities.

Although a large body of works have been proposed with taint tracking (Rawat et al. 2017; Chen and Chen 2018; Jain et al. 2018; Gan et al. 2020), symbolic execution (Cha et al. 2015; Stephens et al. 2016; Yun et al. 2018) and other advanced program analysis techniques (You et al. 2017; Lemieux et al. 2018; Aschermann et al. 2019; You et al. 2019; Wang et al. 2019a), these works are concentrated on generating the well-formed seed inputs (Woo et al. 2013; Wang et al. 2017; Godefroid et al. 2017), selecting the most promising seed inputs (Rebert et al. 2014; Böhme et al. 2016; Gan et al. 2018; Lemieux and Sen 2018), and improving performance and speed (Xu et al. 2017; Zhang et al. 2018; Peng et al. 2018).

More importantly, the effectiveness of fuzzing largely depends on mutation strategy, which is a predetermined set of mutation operators to characterize where and how to mutate seed inputs (Lyu et al. 2019). Fuzzers (e.g., AFL, PTfuzz) then utilize uniform scheduling policy to choose mutation operators at random from this predetermined set, which decreases the overall efficiency of fuzzing. Thus, two critical problems about mutation strategies remain unsolved.

First, *how to efficiently choose the optimal mutation operators for fuzzing?* Given the same applications, the number of crashes and paths found may vary significantly depending on the mutation operators selected. Prior research (Karamcheti et al. 2018; Lyu et al. 2019), for example, illustrated that uniform operators scheduling policy is likely to spend unnecessary computing resources on inefficient operators and reduce the overall fuzzing efficiency. To improve the efficiency of mutation, the key challenge is how to choose optimal mutation operators in each round that maximizes the number of interesting seeds yielded. While several efforts (Böttinger et al. 2018) focus on improving seed mutation, performance improvements are limited on the basis of their experimental results. Thus, a more effective mutation operator selection strategy is crucial to improve the performance of fuzzing (e.g., code coverage and the number of unique crashes).

Second, *how to perform context-aware adaptive mutation?* Apparently, the number of found crashes and paths given the same operators and scheduling policy may also vary with target programs and seed files used. For instance, the initial seeds are mutated by using an operator, which may yield interesting seeds on one program but fail on another one. Recent studies (Karamcheti et al. 2018; Lyu et al. 2019) leverage multi-armed bandit (MAB) algorithm and customized particle swarm optimization (PSO) algorithm to find the optimal probability distribution of mutation operators. Nevertheless, they ignore the characteristics of various seed files, which fail to select proper mutation operators for a specific seed file. Therefore, a context-aware adaptive mutation strategy is essential as it performs pertinent mutations.

In this paper, we proposed a novel context-aware adaptive mutation scheme CMFuzz, which utilizes a contextual bandit algorithm (i.e., LinUCB) to effectively choose optimal mutation operators for various seed files, to address the aforementioned questions. Inspired by the recommender system, we frame fuzzing mutation as a multi-armed bandit problem, in which each arm corresponds to a mutation operator, and the selection of arms in each round is consistent with the selection of mutation operators. Unlike the prior mutation-based fuzzers, our context-aware adaptive mutation strategy enables CMFuzz to dynamically choose potential mutation operators on the basis of the file characteristics. To achieve the goal of context-aware yet low-overhead adaptive mutation, we utilize the byte stream of a seed file to extract the file characteristics for selecting mutation operators. Since byte stream can deliver common but simple content and type information of seed file, we are able to perform high-efficiency mutation based on the specific seed file in fuzzing.

In particular, CMFuzz is a generic approach that can be extended to other state-of-the-art mutation-based fuzzers such as AFLFast (Böhme et al. 2016), FairFuzz (Lemieux and Sen 2018), and VUzzer (Rawat et al. 2017). We have applied it to three advanced fuzzers, i.e., PTfuzz (Zhang et al. 2018), AFL (Zalewski 2019) and AFLFast (Böhme et al. 2016), and implement CMFuzz-PT, CMFuzz-AFL, and CMFuzz-AFLFast, respectively. Furthermore, we evaluated these prototypes on LAVA-M dataset (Dolan-Gavitt et al. 2016) and 12 real-world applications to examine the relative effectiveness of CMFuzz against mainstream fuzzers. The results are very encouraging. CMFuzz explored around 2.59× more unique crashes, 1.66× deeper paths, and 1.43× more unique paths than PTfuzz on these applications.

Additionally, contextual bandit algorithm LinUCB makes CMFuzz much more powerful than other outstanding context-free bandit algorithms, e.g., Thompson Sample and epsilon-greedy. To better examine the efficiency of proposed model, we also employ these bandit algorithms to realize analogous mutation strategy in fuzzing. To this end, we implement Thompson-PT and Greedy-PT based on PTfuzz and evaluate these fuzzers on LAVA-M dataset and 12 real-world applications. The results demonstrated that CMFuzz-PT outperforms Thompson-PT and Greedy-PT (i.e., found 1.79× and 1.11×more crashes than Thompson-PT and Greedy-PT).

In summary, this paper makes the following contributions:

- *Presentation of scheme*: we proposed a novel context-aware adaptive mutation scheme CMFuzz, which utilizes a contextual bandit algorithm to effectively choose optimal mutation operators. It can be extended to masses of existing mutation-based fuzzers.
- *Compatibility of scheme*: we applied CMFuzz to three state-of-the-art fuzzers, which covers PTfuzz, AFL, and AFLFast, and implemented CMFuzz-PT, CMFuzz-AFL, and CMFuzz-AFLFast. We evaluated these prototypes on LAVA-M dataset and 12 widely-used real-world programs, showing that CMFuzz outperforms various advanced fuzzers.
- *Evaluation of scheme*: we also implemented Thompson-PT and Greedy-PT based on PTfuzz using Thompson Sample and epsilon-greedy algorithm. The evaluation illustrated that CMFuzz-PT could discover more unique crashes and paths compared to Thompson-PT and Greedy-PT.

The remainder of this paper is organized as follows. Section 2 clarifies the background and summarizes existing bandit algorithms. Section 3 analyzes the motivation example and depicts the design. Section 4 presents implementation of CMFuzz. In Section 5, we describe various experiments conducted for evaluating the unique paths and crashes of CMFuzz against excellent fuzzers on real-world programs and LAVA-M. Section 6 discusses CMFuzz's limitations and future work. Section 7 presents the related works, and Section 8 offers the conclusion.

2 Preliminaries

In this section, we provide background on mutation-based fuzzers and relevant multi-armed bandit algorithms.

2.1 Mutation-based Fuzzing

Mutation-based fuzzing (Stephens et al. 2016; Rawat et al. 2017; Chen and Chen 2018; Jain et al. 2018; Yun et al. 2018; Gan et al. 2020; Zalewski 2019) often incrementally generates new seeds via modifying well-formed seed inputs employing mutation operators, which shows very promising results in security testing. The general workflow of mutation-based fuzzing is as illustrated in Fig. 1, consisting of stages like seed selection, seed mutation and seed screening. More specifically, (1) the fuzzer receives the initial seeds and maintains a seed queue. (2) *Seed selection* picks a seed from seed queue. (3) *Seed mutation* applies mutation operators on the selected seed to yield various new seeds. (4) *Seed screening* filters new yielded seeds and only sends interesting seeds to seed queue. (5) the fuzzer goes back to step (2) and continues to loop. In this paper, we primary focus on the step (3), i.e., seed mutation.

2.2 Seed Mutation

A good seed mutation strategy critically affects the effectiveness of fuzzing. Mutation-based fuzzers usually apply different seed mutation strategies to yield interesting seeds that can trigger new program paths. Different fuzzers have different mutation strategies and operators (Lyu et al. 2019). Here we take PTfuzz (Zhang et al. 2018) as an example to introduce its mutation strategies. The mutation operators predetermined by PTfuzz are as showed in Table 1.

PTfuzz, as the descendant of AFL, also utilizes deterministic stage and non-deterministic stage to mutate seed inputs. The former is activated for the first time, which adopts 4 types of mutation operators to mutate seeds in sequence. After the former is completed, the latter starts to work, which is a completely randomized process, including havoc stage and splice stage. Splice stage cuts two different seeds out to form a new seed, which is only executed if the other two stages fail to yield interesting seed in one round. In particular, havoc stage randomly employs 9 types of mutation operators to mutate seeds, which is widely used in mutation-based fuzzers. Thus, this paper primarily optimizes the mutation strategy at the havoc stage.

2.3 Bandit Algorithm

As a special reinforcement learning algorithm, multi-armed bandit algorithms are simple but powerful that make decisions over time under uncertain situations (Slivkins 2019). However, the fundamental challenge in bandit algorithms is how to approach the exploration-exploitation trade-off. Exploration selects a seemingly suboptimal arm that might gain better rewards in the future to gather more information, while exploitation chooses the arm that appears best given past experience information. Here we only briefly introduce several effective and frequently-used algorithms, including epsilon-greedy (McCaffrey 2019), Thompson sample (Agrawal and Goyal 2011), and LinUCB (Slivkins 2019).

Epsilon-Greedy The epsilon-greedy algorithm is one of the simplest and forthright algorithms, which occurs in several areas of machine learning. One basic exploration policy of epsilon-greedy is to uniformly select an arm at random with probability ε (exploration), but select the "greedy" arm to perform the exploitation policy with the remaining probability, i.e., 1-ε (exploitation).

Thompson Sample Compared to epsilon-greedy, the exploration policy of Thompson sample is more sophisticated but smarter, which frames the exploration-exploitation tradeoff as a Bayesian posterior estimation. In each trial, this algorithm selects the arm with the highest probability that comes from the samples of Beta/Bernoulli distribution function, which seems better than the "greedy" choice.

Listing 1 Sample code snippet from pngread.c

LinUCB The LinUCB (i.e., Linear Upper Confidence Bound) quite fundamentally differs from the aforementioned context-free bandit algorithms. In contrast, LinUCB is a contextual linear multi-armed bandit algorithm (Li et al. 2010), which can select proper arms for users based on their information. More specifically, LinUCB sequentially selects arms to serve users using the feature vector that summarizes user information or user profiles. This feature vector also can be referred to as the context, which makes LinUCB much more outstanding than context-free bandit algorithms because it can customize personalized services instead of a unified service for diverse users. Moreover, unlike the unguided exploration of epsilon-greedy, exploration in LinUCB algorithm is effectively guided via a confidence interval. In what follows, we give a formal definition of contextual LinUCB.

For each round $t = 1, 2, 3, \ldots$, LinUCB picks an arm a_t and receives a reward r_t by observing their feature vector X_t with dimensions of $m \times 1$, where the vector X_t summarizes the contextual information. Unlike the context-free algorithm, the reward r_t of contextual algorithm in each round t depends both on the context X_t and the chosen arm a_t. Particularly, LinUCB assumes the expected reward $E_{t,a}$ of an arm a is linear in its

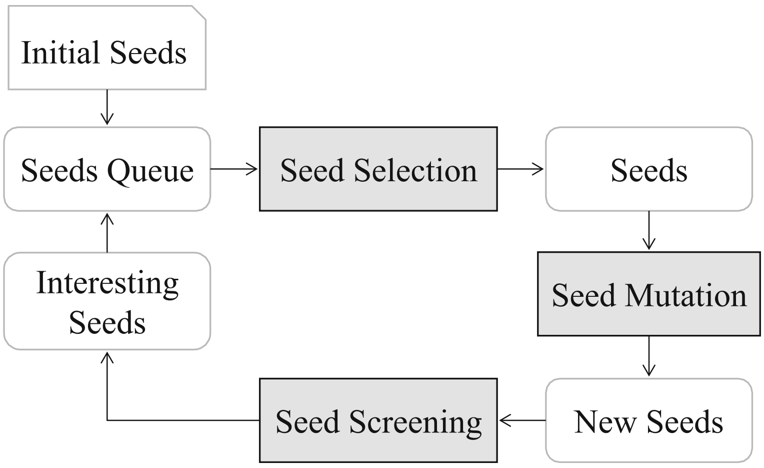


Fig. 1 The workflow of mutation-based fuzzing

Table 1 Mutation operators predetermined by PTfuzz

Stages	Type	Operators
Deterministic stage	bitflip	bitflip 1/1, bitflip 2/1, bitflip 4/1, bitflip 8/8, bitflip 16/8, bitflip 32/8
	arithmetic	arith 8/8, arith 16/8, arith 32/8
	interest	interest 8/8, interest 16/8, interest 32/8
	dictionary	user extras (overwrite), user extras (insert), auto extras (overwrite)
Havoc stage	bitflip	bitflip bit
	interest	interest byte, interest word, interest dword
	addition	addition byte, addition word, addition dword
	subtraction	subtraction byte, subtraction word, subtraction dword
	insertion	insert byte(s)
	deletion	delete byte(s)
	cloning	clone byte(s)
	overwrite	replace byte(s)
	dictionary	extras (overwrite), extras (insert)
Splice stage	cross over	cross over

context $x_{t,\,a}$ (i.e., $m \times 1$) with coefficient vector θ_a^*, as shown in Eq. (1).

$$E_{t,a} = x_{t,a}^T \theta_a^* \tag{1}$$

Where coefficient vector θ_a^* can be estimated by applying existing ridge regression as given in Eq. (2), where A_a is the identity matrix with dimensions of $m \times m$ and B_a is the $m \times 1$ zero vector.

$$\widehat{\theta}_a = (A_a)^{-1} B_a \tag{2}$$

In each round, LinUCB always tries to choose the arm with highest upper confidence bound (i.e., UCB) as the optimal arm, as shown in Eq. (3).

$$p_{t,a} = x_{t,a}^T \widehat{\theta}_a + \alpha \sqrt{x_{t,a}^T A_a^{-1} x_{t,a}} \tag{3}$$

Where $x_{t,a}^T \widehat{\theta}_a$ is the expected reward, α is a constant, and $\sqrt{x_{t,a}^T A_a^{-1} x_{t,a}}$ becomes the standard deviation. After choosing the optimal arm a_t, LinUCB observes a real-valued reward $r'_{t,a}$ and updates the current parameters $A_{t,\,a}$, $B_{t,\,a}$ at trial t to perform subsequent rounds (i.e., $t+1$), as shown in Eq. (4) and Eq. (5).

$$A_{t+1,a} = A_{t,a} + x_{t,a} x_{t,a}^T \tag{4}$$

$$B_{t+1,a} = B_{t,a} + r'_{t,a} x_{t,a} \tag{5}$$

```
1   #define PNG_COLOR_MASK_COLOR    2
2   #define PNG_COLOR_TYPE_RGB    2
3   #define PNG_COLOR_TYPE_RGB_ALPHA    2 | 4
4
5   // Read png pixel data for file and undo intrapixel differencing
6   static void png_do_read_intrapixel(png_row_infop row_info, png_bytep row) {
7       if ((row_info->color_type & PNG_COLOR_MASK_COLOR) != 0) {
8           int bytes_per_pixel;
9           png_uint_32 row_width = row_info->width;
10          if (row_info->bit_depth == 8) {
11              png_bytep rp;
12              png_uint_32 i;
13              if (row_info->color_type == PNG_COLOR_TYPE_RGB)
14                  bytes_per_pixel = 3;
15              else if (row_info->color_type == PNG_COLOR_TYPE_RGB_ALPHA)
16                  bytes_per_pixel = 4;
17              else return; …
18          }
19          else if (row_info->bit_depth == 16) {…
20              if (row_info->color_type == PNG_COLOR_TYPE_RGB)
21                  bytes_per_pixel = 6;
22              else if (row_info->color_type == PNG_COLOR_TYPE_RGB_ALPHA)
23                  bytes_per_pixel = 8;
24              else return; …
25          } …
26      } …
27  }
```

3 CMFuzz

In this section, we illustrate the challenge of current fuzzing techniques by means of a motivating example and a simple experiment. Moreover, we present the overview and details of our context-aware scheme CMFuzz.

3.1 Motivation

We use a simple code snippet shown in Listing 1 as a motivating example to elaborate why a good mutation operator selection strategy is critical for fuzzing. This example code based on the function *png_do_read_intrapixel*() of the libpng library (libpng 2019), which can read and parse the PNG file format. A PNG file includes a series of critical chunks and ancillary chunks, in which the first critical chunk IHDR contains image's width, height, bit depth, color type, compression method, filter method, and interlace method.

In Listing 1, the code snippet demonstrates a general switch-like code pattern, which checks the values of *bit_depth* (i.e., 8 or 16) and *color_type* (i.e., 2 or 4) to set different *bytes_per_pixel*. If the fuzzer empirically prefers bitflip or arithmetic mutation operators such as addition or subtraction, it will, with high likelihood, meet the conditions of *bit_depth* as well as *color_type* and then cover and test more program branches or paths (e.g., line 14 and line 16). More specifically, an effective scheduling strategy would manage to prioritize potential mutation operators like bitflip or arithmetic on the basis of previous selection experience, and de-prioritize other mutation operators that seem to be invalid, thus it would improve the likelihood of generating interesting seeds. Instead, a uniform mutation operator selection strategy such as PTfuzz would likely select all mutation operators with the same probability, which means that the likelihood of selecting effective bitflip of arithmetic mutation operators is almost equal to that of other mutation operators that appear to be ineffective, thus it could be hard to traverse more program paths.

In view of the above example code, it is clear that diverse mutation operator scheduling strategies may produce different efficiency. To further verify mutation strategy, we also fuzz pngfix within libpng library using original uniform scheduling strategy in PTfuzz and adaptive scheduling strategy to evaluate the number of invocations of each mutation operator during the fuzzing. Figure 2a and b show the percentage of 16 mutation operators used in the havoc stage within 2 h. In a uniform scheduling strategy, the proportion of invocations for each mutation operator is about 6%. On the contrary, more than 50% and 20% of mutations are effective arithmetic and bitflip mutation operators in an adaptive scheduling strategy. Other mutation operators such as extras insert and delete byte(s) are less than 4%. Obviously, this adaptive scheduling strategy is more effective than the uniform scheduling strategy in PTfuzz.

Similarly, the invocations of mutation operators may vary with different programs that parse various file format (e.g. tiff, gif, binary) in the light of the analysis for other file formats. Hence, we use another program (i.e., objdump) within GNU binutils that parse binaries to conduct experiments, as shown in Fig. 2c and d. We found that uniform scheduling strategy on objdump is similar to the result of pngfix. Nevertheless, extras overwrite in the adaptive scheduling strategy reaches more than 20%, which is quite different from 1.68% on pngfix. This demonstrated that adaptive scheduling strategy can choose diverse mutation operators for different programs, instead of selecting the same operators for all programs. Therefore, a better adaptive mutation strategy that continually chooses effective mutation operators for diverse program and seed file formats may also be necessary to improve the efficiency of fuzzing and increase the probability of generating interesting seeds that cover more program paths.

3.2 Overview of CMFuzz

In this subsection, we briefly introduce the workflow of our context-aware adaptive mutation scheme CMFuzz. Figure 3 presents a high level overview of CMFuzz. It consists of three major components: file execution, feature extraction, and context-aware mutation.

File Execution CMFuzz as a mutation-based fuzzer, usually receives some initial seeds to start fuzzing and maintain a seed queue to append new seeds during fuzzing mutation. These initial seeds are then used for running target binary program to provide runtime information for context-aware mutation. Inspired by PTfuzz (Zhang et al. 2018), CMFuzz also utilizes Processor Trace mechanism to collect the runtime information during program execution.

Feature Extraction To perform context-aware adaptive mutation strategy, CMFuzz begins by extracting file features from a loaded seed file as contextual information. Specifically, CMFuzz obtains the common byte stream of a seed file as raw file feature for various file format. In addition, to efficiently perform contextual bandit algorithm, the key feature of this raw file feature needs to be further extracted via a matrix decomposition technique to generate a lowdimensional and effective feature vector that denotes context. This feature vector is then used in the next component of CMFuzz, i.e., context-aware mutation.

Context-Aware Mutation Mutation-based fuzzers will mutate seeds in a predefined way to generate new seeds. CMFuzz utilizes contextual bandit algorithm LinUCB to designate which mutation operators should be selected in each round of fuzzing. More precisely, CMFuzz first considers the file feature for a given seed file. Next, it adaptively chooses optimal mutation operators for specific file format using aforementioned feature vector to achieve context-aware mutation. The intuition is that by focusing on the contextual characteristic of seed file, we can devise an adaptive mutation strategy to maximize code coverage. CMFuzz's mutation strategy differs depending on the characteristic of seed file.

3.3 Features

Before elaborating on the context-aware mutation strategy, which is based on the contextual bandit algorithm LinUCB, we first introduce the pivotal context, i.e., the features of seed file.

In this paper, we identify the common byte stream of a seed file as raw feature to achieve high-efficiency and context-aware adaptive mutation. A byte stream is an ordered sequence of bytes in a file. Intuitively, it is a stream of characters in bytes, which can represent the raw content and type information of different seed files with a specific format such as a text file or an image. We choose byte stream primarily for the following reasons. First, the byte stream is

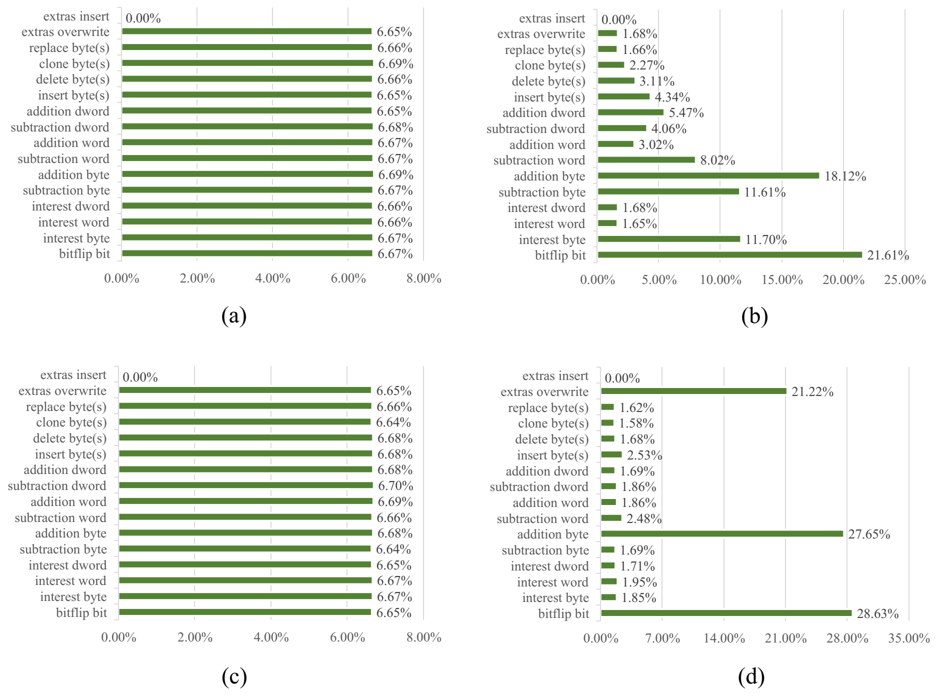


Fig. 2 Percentage of mutation operators invoked during the havoc stage of uniform and adaptive mutation

general enough to map the raw content information of various file formats. Second, it covers the file header identifier indicating the file format information as well. Third, it can be easily obtained without resorting to complex heavyweight program analysis techniques. Thus, this raw feature enables CMFuzz to dynamically choose the optimal mutation operators for diverse seed files.

Although effective and universal, such high-dimensional raw feature may consume computing resources, which affects the efficiency of fuzzing. To avoid the effects of the curse of dimensionality, these raw features thus need to be further extracted to obtain effective feature vectors that denote context for context-aware mutation. CMFuzz employs a previous approach (i.e., non-negative matrix factorization, NMF) (Dhillon and Sra 2005) to extract the key feature and reduce appropriately the dimension of raw file feature. NMF is an effective matrix decomposition method under the constraint that all elements in the matrix are non-negative (Lee and Seung 2001).

Given a non-negative matrix V with dimensions of $n \times m$, NMF tries to approximately factorize into a $n \times r$ basis matrix W and a $r \times m$ coefficient matrix H, where $W \geq 0$, $H \geq 0$, as shown in Eq. (6).

$$V \approx WH \tag{6}$$

Usually r is chosen to be smaller than n or m, so that W and H are smaller than the original matrix V. This results in a compressed version of the original matrix V. Since this compressed matrix approximates the original matrix as much as possible, NMF could capture the underlying structure in the raw data (Lee and Seung 2001).

To measure the quality of the approximation, NMF defines cost function to obtain the approximation of W and H, as shown in Eq. (7). On this basis, W and H can be further computed using multiplicative update rules, as shown in Eq. (8) and Eq. (9).

$$\|V - WH\|^2 = \sum_{ij} \left[V_{ij} - (WH)_{ij} \right]^2 \tag{7}$$

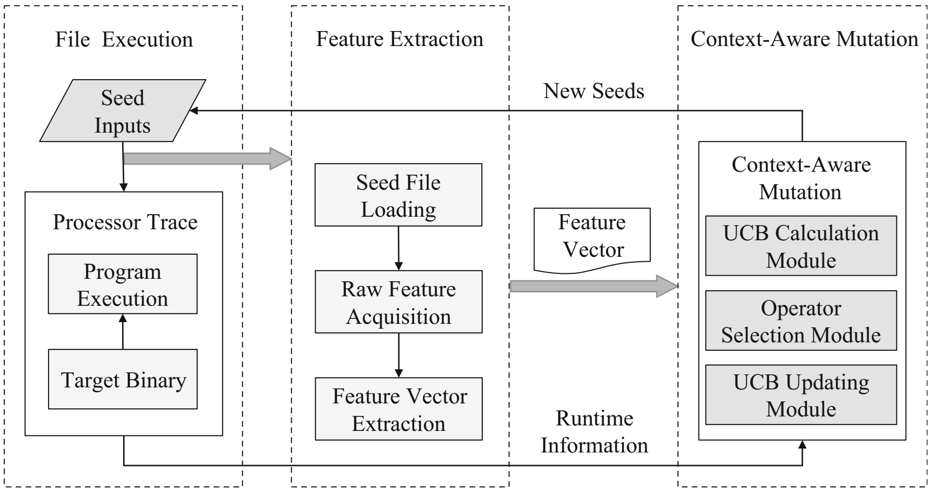


Fig. 3 An overview of CMFuzz

$$W_{ik} \leftarrow W_{ik} \frac{\left(VH^T\right)_{ik}}{\left(WHH^T\right)_{ik}} \tag{8}$$

$$H_{kj} \leftarrow H_{kj} \frac{\left(W^T V\right)_{kj}}{\left(W^T WH\right)_{kj}} \tag{9}$$

Compared with other decomposition methods, the non-negativity constraints of NMF makes it learns part-based representation, which is compatible with the intuitive notion of combining parts to form a whole (Lee and Seung 1999). Hence, NMF could extract the latent local features of byte stream as the final feature vector to perform context-aware mutation.

3.4 Context-Aware Mutation

Algorithm 1 shows the context-aware seed mutation strategy of CMFuzz. We frame fuzzing mutation as a multi-armed bandit problem, in which each arm corresponds to a mutation operator (i.e., the 16 operators of havoc stage in Table 1), the contextual features of users are converted into file features, and the selection of arms in each round is consistent with the selection of mutation operators. Specifically, CMFuzz leverages a contextual bandit algorithm LinUCB to choose optimal mutation operators which are more likely to trigger new coverage. Like other mutation-based fuzzers, CMFuzz also maintains a seed queue that manages to exercise different program paths. After selecting a seed from seed queue (*select_seeds*), CMFuzz extracts the characteristics of seed (*extract_feature*) and encodes them as the context $x_{t,\,a}$. CMFuzz then enters the havoc mutation loop of fuzzing within the range of energy values determined by the power scheduling (*energy*). During each havoc loop, there are two inner loops on lines 5 and 8 in Algorithm 1, with the first loop (line 5) responsible for evaluating the expected reward and upper confidence bound (*calculate_ucb*) for each mutation operator according to Eq. (1) and Eq. (3). The second loop (line 8) is used for mutating the chosen seed s to generate new seed (*havoc_mutation*) by choosing a specific number (*operators_num*) of mutation operator (*select_operator*) on the basis of the value of upper confidence bound. CMFuzz then utilizes new seed to execute target binary program (*execution*) to update the real-valued reward of corresponding mutation operator (*update_reward*) as well as relevant parameters on the basis of Eq. (4) and Eq. (5). In addition, CMFuzz also monitors if it has new coverage (*has_new_coverage*). If it causes a new coverage, then this seed is added to the seed queue (*append*).

4 Implementation

We implemented a prototype of CMFuzz built on top of PTfuzz (Zhang et al. 2018). We chose PTfuzz since it is a state-of-the-art mutation-based fuzzer that records accurately execution information utilizing Processor Trace mechanism. CMFuzz consists of three core components: file execution, feature extraction, and context-aware mutation.

4.1 Feature Extraction Module

To achieve context-aware yet low-overhead adaptive mutation strategy, this module first sets the common byte stream of loaded seed file as raw file feature, which makes CMFuzz not limited to a specific file format. Additionally, CMFuzz treats the byte stream of seed file as a byte array and groups every eight bytes of a byte array into a byte matrix to facilitate intuitive understanding and subsequent processing. Then, CMFuzz further utilizes NMF to extract key feature and reduce the dimension of raw file feature. More specifically, we first construct the original data matrix V through performing column vectorization on several files, and compute the basis matrix W on the basis of Eq. (8) and Eq. (9) to minimize Eq. (7) until it is stable. We then use computed basis matrix W to map the coefficient matrix H for the column vectorized byte matrix. This coefficient matrix H is the final feature vectors of seed file, which reflects the latent local characteristics of raw data like the same file header identifier or other similar content information.

Here we determine a relative dimension (i.e., m = 8) through initial experimental verification and set $r = 1$. After dimensionality reduction, we obtain an eight-dimensional feature vector that can represent the contextual characteristics of seed file.

Algorithm 1 Context-Aware Mutation

```
Require: Target binary program: $P$
         Seed queue: $Q$
         Mutation operators: $M$
1   while In the fuzzing loop do
2     $s = select\_seeds\ (Q)$
3     $v = extract\_feature\ (s)$
4     for $i$ from 0 to $energy$ (s) do
5         for each $m \in M$ do
6             $ucb_m = calculate\_ucb\ (v, m)$
7         end for
8         for $j$ from 0 to $operators\_num$ () do
9             $op_j = select\_operator\ (ucb, j)$
10            $seed = havoc\_mutation\ (s, op_j)$
11        end for
12        $coverage = execution\ (P, seed)$
13        $update\_reward\ (reward, coverage, M)$
14        if $has\_new\_coverage\ (seed)$ then
15            $Q = append\ (Q, seed)$
16        end if
17    end for
18  end while
```

4.2 Context-Aware Mutation Module

The context-aware mutation module consists of three major submodules: the UCB calculation module, mutation operator selection module, and UCB updating module.

UCB Calculation Module To choose the optimal mutation operator, this module first initializes several parameters of the LinUCB algorithm to calculate the upper confidence bound (i.e., UCB) of each mutation operator. More specifically, CMFuzz initializes the parameter A_a of each operator with an identity matrix, and sets the initial metadata feature vector B_a of each operator to zero vector. The upper confidence bound of each mutation operator thus can be calculated on the basis of parameter A_a and B_a, following Eq. (1), Eq. (2), and Eq. (3).

Operator Selection Module This module selects the optimal mutation operator to perform fuzzing mutation by considering the highest upper confidence bound of each operator. Note that, to alleviate the credit assignment problem, we reference the work of Karamcheti et al. (2018) to set a small constant (i.e., 4) as the number of mutation operators, which corresponds to the second inner loop (line 8) in Algorithm 1. Hence, the criterion for mutation operator selection becomes to choose the first four operators with the highest upper confidence bound among all mutation operators in order.

UCB Updating Module With the runtime information provided by the file execution module, this module updates the reward and parameter A_a and B_a of selected mutation operators in a round by applying Eq. (4) and Eq. (5). Then, we can further recalculate their upper confidence bound using updated reward and parameter A_a and B_a. After updating each parameter of the selected mutation operator, CMFuzz will enter the next round of fuzzing.

5 Evaluation

In this section, we proceed to evaluate the efficiency and effectiveness of CMFuzz on the LAVA-M dataset and 12 open source real-world applications by comparing CMFuzz with other state-of-the-art fuzzers.

5.1 Experimental Setup

Experiment environment We conduct all the following experiments on a machine running 64-bit Ubuntu 16.04 LTS equipped with one Intel CPU @3.70GHz and 8GB RAM.

Baseline fuzzers We select several state-of-the-art mutation-based fuzzers, including PTfuzz (Zhang et al. 2018), AFL (Zalewski 2019), AFLFast (Böhme et al. 2016), and MOPT (Lyu et al. 2019), as our benchmarks to examine the performance of proposed CMFuzz. Furthermore, we implement the prototypes of CMFuzz-PT, CMFuzz-AFL, and CMFuzz-AFLFast based on their counterparts, respectively. In particular, we also implement Thompson-PT and Greedy-PT based on PTfuzz employing Thompson Sample and epsilon-greedy algorithm, respectively.

Evaluation metrics We preferentially employ three major evaluation metrics, namely unique crashes, path depth, and path coverage, to compare the efficiency of each fuzzer with CMFuzz from diverse perspectives. For the first metric, we count the number of unique crashes found to evaluate the crash discovery capabilities of various fuzzers. Similarly, we summarize the maximum path depth in fuzzing and the number of unique paths discovered by different fuzzers to measure the path discovery capabilities of fuzzers. Furthermore, we track the growth trend of unique paths as well.

5.2 Datasets

We choose 12 popular open source real-world Linux applications to evaluate CMFuzz as shown in Table 2, each of which has different functionality receiving a wide range of input format, including well-known GNU binutils (e.g., strings, size, nm), tiff processing libraries (e.g., libtiff), png processing libraries (e.g., libpng), text formatting processing program (e.g., w3m), and other image processing program (e.g., gif2png, jhead). These real-world programs are also broadly used to evaluate other fuzzers (Böhme et al. 2016; Zhang et al. 2018; Jain et al. 2018; Gan et al. 2020; Wang et al. 2019a; Lyu et al. 2019). In addition, we employ the LAVA-M dataset (Dolan-Gavitt et al. 2016) for evaluation, which consists of four GNU coreutils program (i.e., who, uniq, base64, and md5sum).

5.3 Effectiveness Evaluation

To show how effectively our context-aware mutation scheme can improve the performance fuzzing, we measure the achieved unique crashes, path depth, and code coverage during the fuzzing process by using CMFuzz and PTfuzz on the 12 frequently-used applications in Table 2 with the same seed file. Here we conduct three trials for each item to avoid randomness, with each trial runs for 24 h. Table 3 shows the average value of unique paths, maximum depth, and unique crashes found by CMFuzz and PTfuzz. We can infer the following conclusions from Table 3.

Unique Paths Form Table 3, we see that CMFuzz significantly outperforms PTfuzz in all applications. For example, the average unique paths found by CMFuzz increase almost 339.71% and 67.18% over PTfuzz on infotocap and size, respectively. Among these 12 applications, CMFuzz finds 10,850 unique paths in total, which increases by 42.63% over PTfuzz. The results clearly indicate that our proposed CMFuzz has much better performance than PTfuzz in discovering unique paths.

Maximum Depth In terms of the maximum depth, CMFuzz can explore deeper paths than PTfuzz in all applications as well, generally showing improvements in maximum depth. More importantly, CMFuzz outperforms PTfuzz at most 600.00% in these 12 applications. In total, CMFuzz explores 231 more maximum depth than PTfuzz, which is around 66.19%. Therefore, CMFuzz can improve the maximum path depth of fuzzing observably.

Unique Crashes CMFuzz also outperforms PTfuzz on most cases in finding unique crashes, on average by 159.18%. Especially in jhead, infotocap, and pngfix, only CMFuzz reports the unique crashes yet PTfuzz fails. CMFuzz finds 17, 43, and 27 unique crashes in strings, nm, and tiff2pdf where PTfuzz only finds 6, 13, and 13 unique crashes, which is around 183.33%, 230.77%, and 107.69% more than PTfuzz, respectively. Overall, CMFuzz discovers 127 more unique crashes than PTfuzz on these 12 applications. Particularly, these unique crashes discovered by CMFuzz basically subsumed those of PTfuzz, and new bugs were found in several programs like jhead. We have reported these new bugs to their developers.

To further demonstrate the effectiveness of CMFuzz, we also tracked the growth trend of number of unique paths explored within 24 h in three repeated experiments and presented in Fig. 4. The y-axis of each plot shows the cumulative number of unique paths and x-axis of plot shows the time. From Fig. 4, we see that CMFuzz basically achieves the upper bound in code coverage on repeated trials of all programs, especially on sam2p, infotocap, size, nm, and strings. More importantly, the unique paths of CMFuzz still has an obvious growth trend at 24 h on size, strings, nm, w3m, and pngfix, which implies that they will reach a higher upper bound if we continue to fuzz. Furthermore, CMFuzz could cover more unique paths in a faster pace than PTfuzz in most applications such as infotocap, sam2p, and tiff2ps.

5.4 Compatibility Evaluation

To evaluate the compatibility of CMFuzz, we also apply our scheme to several other state-of-the-art mutation-based fuzzers such as AFL (Zalewski M 2019) and AFLFast (Böhme et al. 2016), and implement the CMFuzz-AFL as well as CMFuzz-AFLFast based on their counterparts.

We use 12 representative real-world applications in Table 2 as fuzzing target to evaluate CMFuzz-AFL, CMFuzz-AFLFast, and their counterparts. Each experiment is lasted for 24 h with the same settings as in Section 5.1. Table 4 summarizes the number of unique paths, crashes, and path depth found by 6 different fuzzers. Here we discuss the Table 4 from the following three perspectives.

Unique Paths CMFuzz-based fuzzers have much better performance than their counterparts (i.e., PTfuzz, AFL, and AFLFast) in exploring unique paths in most applications. For example, CMFuzz-AFL explores 964 more unique paths than AFL on gif2png, which increases by nearly 246.76%. CMFuzz-AFLFast covers 2649 unique paths on size, which is more than 33.59% increase compared to AFLFast. Overall, CMFuzz-PT, CMFuzz-AFL, and CMFuzz-AFLFast find 10,850, 7476, and 37,359 unique paths, which is around 42.63%, 17.86%, and 18.08% higher than PTfuzz, AFL, and AFLFast, respectively. As a result, CMFuzz is compatible with mutation-based fuzzers and enables these fuzzers to explore more program paths.

Additionally, CMFuzz-AFLFast significantly outperforms other fuzzers in exploring unique paths on all the applications except exiv2 and tiff2ps. For instance, the unique paths found by CMFuzz-AFLFast increase almost 718.15%, 515.42%, and 495.53% over CMFuzz-PT on w3m, pngfix, and sam2p, respectively. CMFuzz-AFLFast explores approximately 764.63%, 542.50%, and 256.43% more unique paths than CMFuzz-AFL on w3m, sam2p, and gif2png, respectively. Overall, the number of unique paths discovered by CMFuzz-AFLFast significantly exceeds CMFuzz-AFL and CMFuzz-PT at almost 399.72% and 244.32%, respectively.

Table 2 Real-world applications used in evaluation

Applications	Project	Version	Input format
strings	binutils	2.23	binary
size	binutils	2.23	binary
nm	binutils	2.23	binary
jhead	jhead	2.97	jpg
sam2p	sam2p	0.49.4	bmp
infotocap	ncurses	6.0	text
gif2png	gif2png	3.0.0	gif
exiv2	exiv2	0.27.2	jpg
pngfix	libpng	1.6.34	png
w3m	w3m	0.5.3	text
tiff2ps	libtiff	3.9.5	tiff
tiff2pdf	libtiff	3.9.5	tiff

Maximum Depth The maximum depth explored by CMFuzz-based fuzzers performs better than their baseline fuzzers on most cases as well. For instance, CMFuzz-AFL explores 118.18% and 111.11% deeper paths than AFL on gif2png and tiff2ps, while the path depth found by CMFuzz-AFLFast increases 371.43% and 200.00% over AFLFast on w3m and pngfix. Overall, CMFuzz-PT, CMFuzz-AFL, and CMFuzz-AFLFast outperforms their baselines by approximately 66.19%, 43.86%, and 70.77% in these 12 applications. Thus, CMFuzz enables various mutation-based fuzzers to explore deeper paths.

Moreover, CMFuzz-PT outperforms other fuzzers on several applications such as strings, size, nm, and tiff2pdf. Especially in nm, the maximum depth of CMFuzz-PT is increased by 540.00% and 255.56% compared with CMFuzz-AFL and CMFuzz-AFLFast. Nevertheless, CMFuzz-AFL explores deeper paths than other fuzzers on infotocap and tiff2ps, while CMFuzz-AFLFast and AFLFast have better performance on sam2p, gif2png, and pngfix.

Unique Crashes CMFuzz-based fuzzers can discover more unique crashes than their counterparts. For example, CMFuzz-AFL finds 19 and 12 more unique crashes than AFL on

Table 3 The unique paths, maximum depth, and unique crashes found by CMFuzz and PTfuzz on 12 applications

Applications	PTfuzz			CMFuzz					
	Paths	Depth	Crashes	Paths	Increase	Depth	Increase	Crashes	Increase
strings	927	14	6	1412	+52.32%	21	+50.00%	17	+183.33%
size	908	12	15	1518	+67.18%	18	+50.00%	28	+86.67%
nm	1263	19	13	1955	+54.79%	32	+68.42%	43	+230.77%
jhead	315	12	0	352	+11.75%	16	+33.33%	1	+1
sam2p	724	13	0	985	+36.05%	17	+30.77%	0	+0.00%
infotocap	68	2	0	299	+339.71%	14	+600.00%	1	+1
gif2png	1198	9	0	1454	+21.37%	28	+211.11%	0	+0.00%
exiv2	50	7	0	54	+8.00%	8	+14.29%	0	+0.00%
pngfix	344	7	0	467	+35.76% =+	14	+100.00%	6	+6
w3m	1496	24	0	1972	+31.82%	33	+37.50%	0	+0.00%
tiff2ps	218	9	2	256	+17.43%	16	+77.78%	4	+100.00%
tiff2pdf	96	11	13	126	+31.25%	14	+27.27%	27	+107.69%
total	7607	139	49	10,850	+42.63%	231	+66.19%	127	+159.18%

infotocap and nm, which increases by 216.67% and 71.43%, respectively. The unique crashes found by CMFuzz-AFLFast on size are approximately 700.00% more than AFLFast. In total, CMFuzz-PT, CMFuzz-AFL, and CMFuzz-AFLFast discover 127, 139, and 85 unique crashes, which is an average 159.18%, 80.52%, and 63.46% increase over PTfuzz, AFL, and AFLFast, respectively. Therefore, CMFuzz can guide different mutation-based fuzzers to trigger more unique crashes as well.

Furthermore, CMFuzz-PT has better performance than other fuzzers on strings, size, nm, jhead, and pngfix. For instance, CMFuzz-PT discovers 258.33% and 750.00% more crashes than CMFuzz-AFL on nm and strings, while nearly 1300.00% and 1600.00% more than CMFuzz-AFLFast. Especially in jhead, only CMFuzz-PT finds 1 unique crash where other fuzzers find no crash at all. However, CMFuzz-AFLFast and AFLFast can find more unique crashes than other fuzzers on infotocap, while CMFuzz-AFL performs better on w3m and tiff2pdf.

5.5 Model Evaluation

To better examine the efficiency of proposed model, we employ other outstanding bandit algorithms, e.g., Thompson Sample and epsilon-greedy, to realize analogous mutation strategy in fuzzing. We also implement Thompson-PT and Greedy-PT based on PTfuzz and evaluate these fuzzers on 12 real-world applications, with each experiment runs for 24 h. The results are shown in Table 5, from which we have the following deduction.

Unique Paths Similar to CMFuzz-PT, Thompson-PT and Greedy-PT also improve the path coverage of PTfuzz in most cases, on average by 10.77% and 36.33%. For instance, the unique paths found by Thompson-PT increase by 51.32% over PTfuzz on size, while Greedy-PT is around 71.59% increase over PTfuzz.

In addition, CMFuzz-PT still performs better than Thompson-PT and Greedy-PT in unique paths for most applications. For example, CMFuzz-PT explores around 225.00% and 18.65% more unique paths than Thompson-PT and Greedy-PT on infotocap, respectively. In total, the number of unique paths found by CMFuzz-PT exceeds Thompson-PT and Greedy-PT nearly 28.77% and 4.62% on these applications.

Maximum Depth For most applications, Thompson-PT and Greedy-PT outperform PTfuzz in the maximum depth, with an average increase of 16.55% and 51.08%. For instance, Thompson-PT and Greedy-PT explore 28.57% and 114.29% deeper than PTfuzz on pngfix, respectively.

Furthermore, CMFuzz-PT also has better performance than Thompson-PT on most programs in the maximum depth, which is an average 42.59% increase. For example, the maximum depth of CMFuzz-PT is over 130.00% on strings compared to Thompson-PT. Although CMFuzz-PT performs slightly worse than Greedy-PT on several programs such as size, sam2p, and pngfix, it surpasses Greedy-PT with a significant advantage on nm and gif2png. Overall, CMFuzz-PT is still slightly better than Greedy-PT in the maximum depth, on average by 10.00%.

Unique Crashes As for the number of unique crashes, Thompson-PT exceeds PTfuzz on size and nm at almost 46.67% and 92.31%, respectively. Greedy-PT performs significantly better than PTfuzz on strings, size, nm, and tiff2pdf, increased by 166.67%, 140.00%, 107.69%, and 146.15%, respectively. The number of unique crashes found by Thompson-PT and Greedy-PT are roughly the same as the crashes of PTfuzz for some programs like jhead, sam2p, and

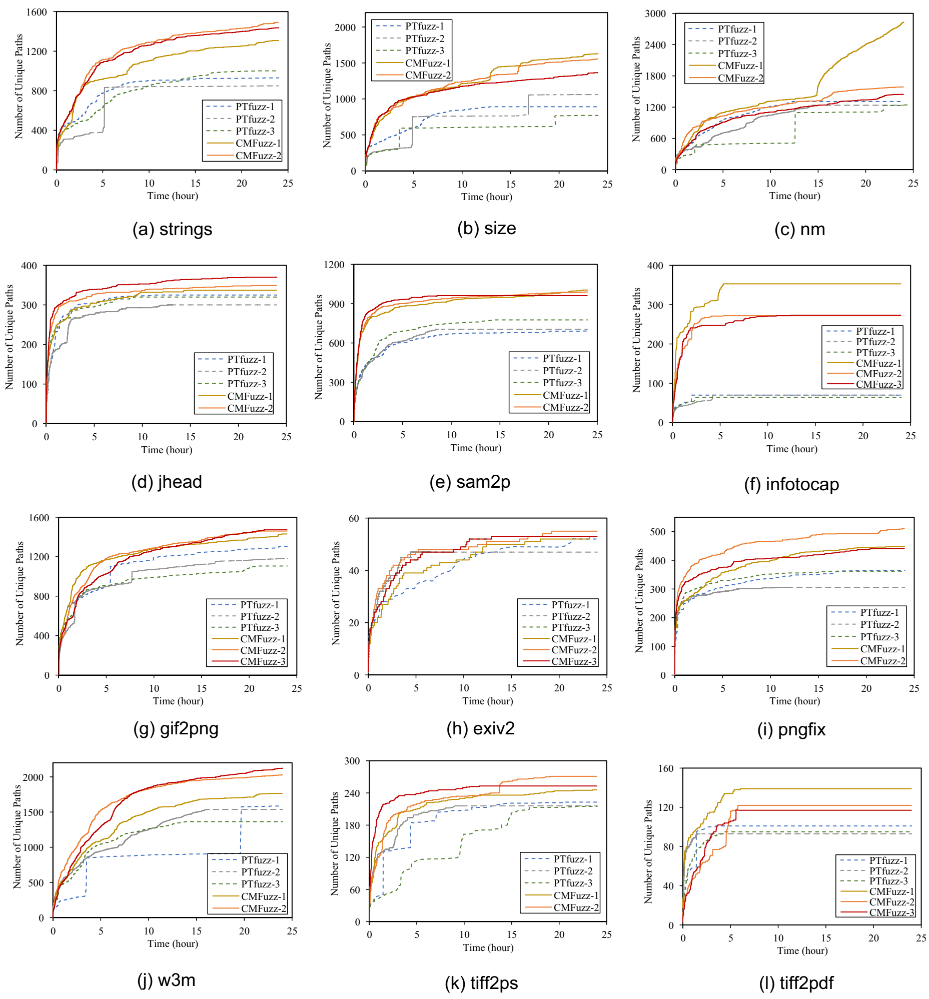


Fig. 4 Number of unique paths discovered by PTfuzz and CMFuzz over time in real-world applications in 24 h

gif2png. In total, Thompson-PT and Greedy-PT find on average 44.90% and 132.65% more crashes than PTfuzz, respectively.

Additionally, CMFuzz-PT can discover more unique crashes than Thompson-PT and Greedy-PT on most cases. For instance, CMFuzz-PT finds 72.00% more crashes than Thompson-PT on nm, and discovers 59.26% more crashes than Greedy-PT. However, CMFuzz-PT, like the other three fuzzers, cannot find unique crashes for sam2p, gif2png, exiv2, and w3m. Overall, the number of unique crashes found by CMFuzz-PT is an average 78.87% and 11.40% increase over Thompson-PT and Greedy-PT, respectively.

5.6 Comparison with MOPT

To further examine the effectiveness of CMFuzz, we also compare CMFuzz with state-of-the-art MOPT (Lyu et al. 2019), a recently introduced fuzzer built on top of AFL, which optimizes

Table 4 Number of unique paths, maximum depth, and crashes found in real-world programs by various fuzzers

Applications	PTfuzz			AFL			AFLFast		
	Paths	Depth	Crashes	Paths	Depth	Crashes	Paths	Depth	Crashes
strings	927	14	6	514	7	2	1732	6	0
size	908	12	15	753	6	10	1983	8	1
nm	1263	19	13	727	5	7	3098	6	1
jhead	315	12	0	292	8	0	924	12	0
sam2p	724	13	0	774	13	0	5279	18	0
infotocap	68	2	0	267	15	6	565	15	31
gif2png	1198	9	0	278	11	0	3182	29	1
exiv2	50	7	0	4	2	0	4	2	0
pngfix	344	7	0	445	7	0	2770	11	0
w3m	1496	24	0	1999	23	47	11,939	7	0
tiff2ps	218	9	2	184	9	3	54	4	5
tiff2pdf	96	11	13	106	8	2	109	12	13
total	7607	139	49	6343	114	77	31,639	130	52
Applications	CMFuzz-PT			CMFuzz-AFL			CMFuzz-AFLFast		
	Paths	Depth	Crashes	Paths	Depth	Crashes	Paths	Depth	Crashes
strings	1412	21	17	720	7	2	1640	7	1
size	1518	18	28	778	6	12	2649	9	8
nm	1955	32	43	765	5	12	2894	9	3
jhead	352	16	1	310	13	0	959	16	0
sam2p	985	17	0	913	19	0	5866	33	0
infotocap	299	14	1	328	19	19	721	15	31
gif2png	1454	28	0	964	24	0	3436	42	5
exiv2	54	8	0	4	2	0	4	3	0
pngfix	467	14	6	486	18	0	2874	33	0
w3m	1972	33	0	1866	24	56	16,134	33	0
tiff2ps	256	16	4	215	19	8	55	9	8
tiff2pdf	126	14	27	127	8	30	127	13	29
total	10,850	231	127	7476	164	139	37,359	222	85

AFL's mutation strategy with an optimization algorithm, i.e., Particle Swarm Optimization (PSO). Each experiment runs for 24 h with the same settings as in Section 5.1.

Figure 5 shows the growth trend of unique paths discovered by MOPT and CMFuzz. Thanks to the context-aware adaptive mutation, CMFuzz shows a strong and stable growth trend in finding unique paths. More specifically, CMFuzz keeps finding unique paths during the later iterations of fuzzing, especially for strings, size, nm, and sam2p. Interesting, MOPT shows a fast growth in the beginning yet reaches a bottleneck after a few hours when testing strings, infotocap, and tiff2pdf. Overall, CMFuzz discovers about 1.4× more unique paths than MOPT.

Furthermore, we also tracked the growth trend of unique crashes and presented in Fig. 6. Note that, we only show the applications where two fuzzers found a crash. As shown in the figure, we could find CMFuzz trigger more unique crashes in a stable pace than MOPT in most subjects like strings and size. For nm and pngfix, CMFuzz could find unique crashes faster, and find much more. Especially, when fuzzing pngfix, CMFuzz find around 19× more crashes in 24 h. However, the number of unique crashes found by CMFuzz is inferior to MOPT for infotocap and gif2png. One possible reason is that our fixed number of mutation operators limits its capabilities, whereas MOPT could choose any number of operators.

5.7 Evaluation on LAVA-M

To further evaluate the effectiveness of proposed CMFuzz, we compare CMFuzz with other mutation-based fuzzers using the standard benchmarks LAVA-M dataset (Dolan-Gavitt et al. 2016) consisting of four buggy programs, who, uniq, base64, and md5sum. LAVA-M is a test suite that manually injects bugs in Linux utilities to evaluate fuzzers. We run PTfuzz, CMFuzz-PT, Thompson-PT, Greedy-PT, AFL, CMFuzz-AFL, AFLFast, and CMFuzz-AFLFast on LAVA-M for 24 h with the same initial seeds and same settings as in Section 5.1. Table 6 shows the cumulative number of unique paths, path depth, and crashes found by these fuzzers on LAVA-M. Although we cannot fully reproduce the original experimental results due to the different configuration of machine and the randomness of fuzzing, the results in Table 6 are still credible under the same configuration and initial seed. Here we also discuss the Table 6 from the following three perspectives.

Unique Paths CMFuzz-based fuzzers have an outstanding performance in comparison to their counterparts in discovering unique paths on LAVA-M. For example, the unique paths explored by CMFuzz-PT and CMFuzz-AFL increase by approximately 195.71% and 40.16% on who compared to PTfuzz and AFL, respectively. Overall, CMFuzz-PT, CMFuzz-AFL, and CMFuzz-AFLFast on average find 68.68%, 9.62%, and 31.66% more paths than their counterparts, respectively.

Furthermore, Thompson-PT and Greedy-PT also significantly outperform PTfuzz yet slightly lower than CMFuzz-PT in finding paths on LAVA-M. For instance, the unique paths of Thompson-PT and Greedy-PT on who are around 101.43% and 52.86% higher than PTfuzz, but approximately 31.88% and 48.31% lower than CMFuzz-PT, respectively. Unfortunately, the number of unique paths for CMFuzz-PT performs slightly lower than Greedy-PT on uniq and md5sum. Overall, CMFuzz-PT, however, slightly exceeds Greedy-PT, on average by 11.64%. In addition, the number of unique paths explored by CMFuzz-PT is an average 13.56% increase over Thompson-PT.

Maximum Depth For most cases, CMFuzz-based fuzzers can explore deeper paths than their baselines as well. CMFuzz-PT and CMFuzz-AFL, for example, explore 171.43% and 216.67% deeper paths than PTfuzz and AFL on who. In total, the maximum depths of CMFuzz-PT, CMFuzz-AFL, and CMFuzz-AFLFast increase by 50.00%, 63.83%, and 40.00% compared to their baselines, respectively.

Moreover, Thompson-PT and Greedy-PT perform better than PTfuzz in terms of exploring path depth as well, on average by 33.33% and 30.56%, respectively. In addition, CMFuzz-PT can explore deeper paths than Thompson-PT and Greedy-PT, especially for who and md5sum.

Unique Crashes In terms of the number of unique crashes, CMFuzz-based fuzzers are a little better than their counterparts on most cases. For example, the unique crashes of CMFuzz-AFL increase by 100.00% on who in comparison to AFL. CMFuzz-PT discovers around 100.00% and 50.00% more crashes than PTfuzz on uniq and md5sum, respectively. In total, the number of unique crashes of CMFuzz-PT and CMFuzz-AFL are almost 50.00% and 118.18% increase over their counterparts on average, respectively. However, CMFuzz-AFLFast is consistent with AFLFast in finding unique crashes on LAVA-M.

Additionally, Thompson-PT and Greedy-PT are slightly inferior to PTfuzz in discovering unique crashes on LAVA-M, on average by 25.00% and 12.50%,

Table 5 Number of unique paths, maximum depth, and crashes found by various bandit algorithm

Applications	PTfuzz			CMFuzz-PT			Thompson-PT			Greedy-PT		
	Paths	Depth	Crashes	Paths	Depth	Crashes	Paths	Depth	Crashes	Paths	Depth	Crashes
strings	927	14	6	1412	21	17	793	9	0	1601	20	16
size	908	12	15	1518	18	28	1374	17	22	1558	20	36
nm	1263	19	13	1955	32	43	1550	19	25	1768	20	27
jhead	315	12	0	352	16	1	358	12	0	354	16	0
sam2p	724	13	0	985	17	0	793	15	0	821	21	0
infotocap	68	2	0	299	14	1	92	8	0	252	14	0
gif2png	1198	9	0	1454	28	0	984	11	0	1408	18	0
exiv2	50	7	0	54	8	0	53	6	0	55	7	0
pngfix	344	7	0	467	14	6	368	9	0	416	15	0
w3m	1496	24	0	1972	33	0	1697	27	0	1747	29	0
tiff2ps	218	9	2	256	16	4	248	19	3	254	19	3
tiff2pdf	96	11	13	126	14	27	116	10	21	137	11	32
total	7607	139	49	10,850	231	127	8426	162	71	10,371	210	114

respectively. Nevertheless, Greedy-PT finds approximately 16.67% more unique crashes than PTfuzz on md5sum. More importantly, CMFuzz-PT outperforms Thompson-PT and Greedy-PT in finding unique crashes, which on average increases by 100.00% and 71.43%, respectively.

5.8 Parameter Sensitivity Analysis

Estimating Dimensions The performance of CMFuzz depends on the choices of different dimensions for feature vector. In Section 4.1, we obtain an eight-dimensional feature vector (i.e., $m = 8$) through dimensionality reduction. To estimate various dimensions, we run multiple trials of PTfuzz for a fixed amount of time, with diverse dimensions m, where the value of m ranges from $\{4, 6, 8, 10, 12\}$. Here we run 2 h for each trial.

The results in Fig. 7 show the mean and standard error of unique paths found across various m every 30 min in 2 h. These statistics were collected through the data sets in Table 2. For m=6, m=8, and m=10, their path statistics have a significant advantage over m=4 and m=12 in 2 h. Since m=8 at 60 Min, 90 Min, and 120 Min perform better than the other two, we choose m=8 for our experiments.

Estimating Operators_Num In Section 4.2, we set a fixed constant for *operators_num* (line 8 of Algorithm 1) to alleviate the credit assignment problem. To evaluate this constant, we reference the method of Karamcheti et al. (2018). Let *num* (i.e., *operators_num*) be the number of mutation operators during the fuzzing. We employ fixed constant *num* to perform several trials of PTfuzz by fuzzing the data sets in Table 2 over a fixed period of time. Note that the range of constant samples is uniformly among $\{2^0, 2^1, 2^2, 2^3, 2^4\}$.

Figure 8 reports the mean and standard error of path statistics across diverse *num* every 30 min in 2 h. Interestingly, except that the mean of *num*=4 is roughly the same as *num*=8 at 90 Min, *num*=4 at 30 Min, 60 Min, and 120 Min are slightly better than *num*=1, *num*=2, *num*=8, and *num*=16. Therefore, we decided to employ *num*=4 to implement our scheme.

6 Discussion

We discuss the experimental results of CMFuzz, using CMFuzz with other mutation-based fuzzers, the limitations of CMFuzz, and future direction of research.

Verification of CMFuzz We evaluate the effectiveness of CMFuzz on 12 real-world applications and LAVA-M dataset. For most applications, CMFuzz-PT is much more efficient in exploring unique paths, maximum depth, and unique crashes than state-of-the-art PTfuzz. Also, CMFuzz-based fuzzers has better performance compared to their counterparts. More importantly, we also compare CMFuzz-PT with other advanced optimization algorithms and bandit algorithms such as PSO and Thompson Sample to further examine the efficiency of proposed model. The experimental results illustrate that CMFuzz-PT outperforms them in discovering unique paths and unique crashes on most cases.

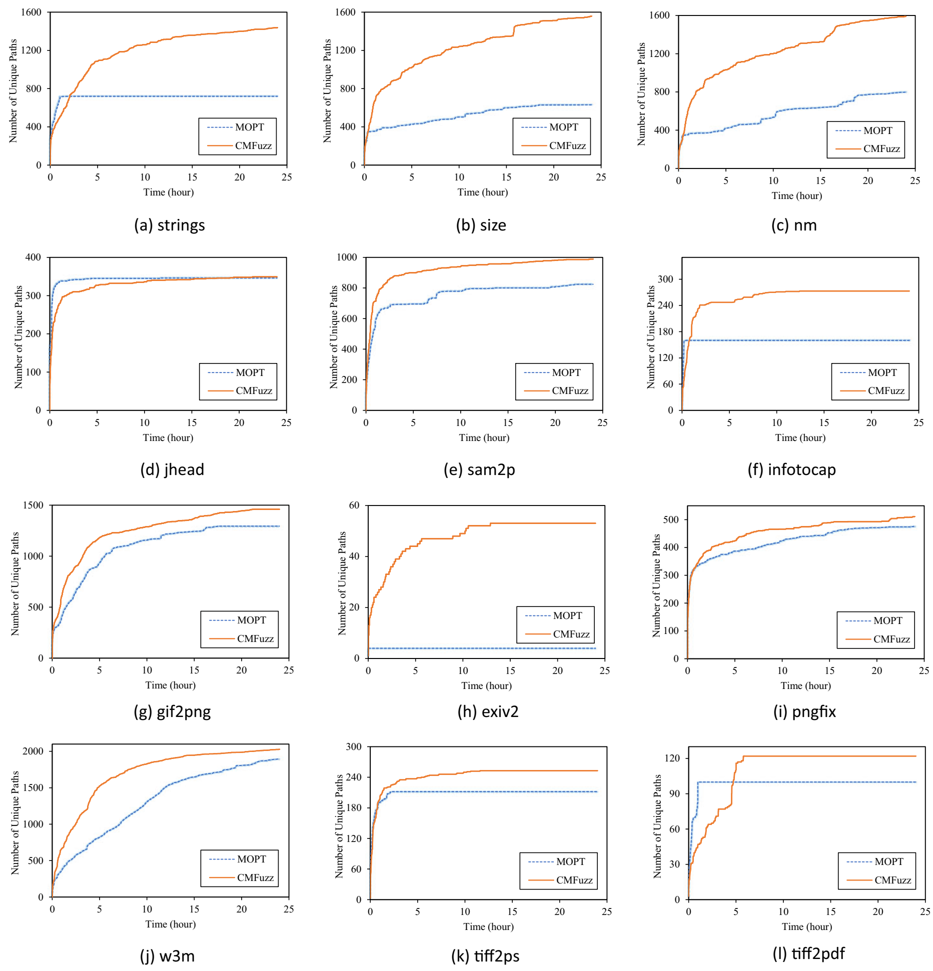


Fig. 5 Number of unique paths found by MOPT and CMFuzz over time in real-world applications in 24 h

Complementing with other Fuzzers Our proposed CMFuzz is a generic scheme, which can be generalized to a wide range of existing mutation-based fuzzers, such as AFL, AFLFast, FairFuzz, and VUzzer. For instance, AFLFast improves the power schedules and prioritizes seeds that are rarely picked, which should be well combined with the proposed context-aware adaptive mutation strategy CMFuzz. In this paper, we apply CMFuzz to PTfuzz, AFL, and AFLFast to elaborate the effectiveness and compatibility of CMFuzz. Additionally, since fuzzing can be modeled as a multi-armed (MAB) problem (Wang et al. 2019a), bandit algorithm such as LinUCB adopted in this paper should also be applicable to other stage of fuzzing, e.g., seed selection stage.

Limitations Although effective, CMFuzz still has a lot to be improved. To achieve low-overhead mutation, we currently select the byte stream of seed file as file characteristics, which could be limited in accuracy. In the future, we will pay more attention to overcome this limitation by extracting other typical as well as refined characteristics (e.g., semantic and syntactic characteristics) for specific seed file to improve the efficiency of fuzzing. Furthermore, other factors, such as credit assignment

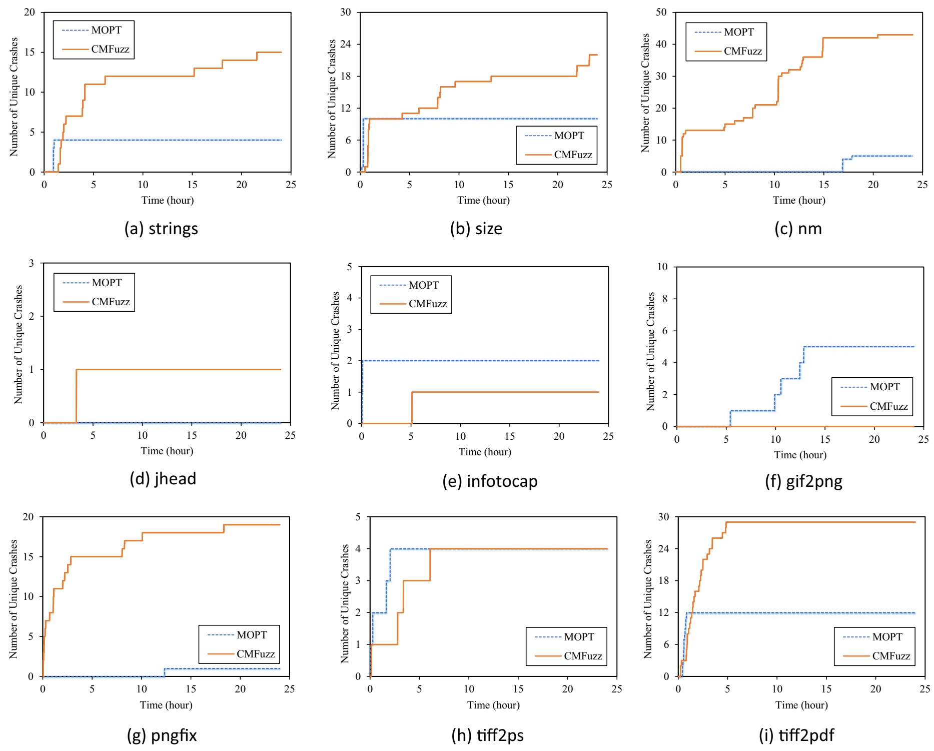


Fig. 6 The growth trend of unique crashes found by MOPT and CMFuzz in 24 h

problem, also may suppress the power of CMFuzz. Currently, we only set four mutation operators to perform mutation in each round. Perhaps any choice of mutation operators could be powerful enough to discover more underlying vulnerabilities. Thus, we will try to get rid of the credit assignment problem in the future.

In addition to the above-mentioned research direction, there are several interesting directions for future work. While the focus of our paper was on applying LinUCB algorithm, it would be worth exploring how to apply hybrid linear model or other contextual bandit algorithm to guide fuzzing. Also, our fuzzing time is not completely enough compared to other works such as MOPT (Lyu et al. 2019) and GREYONE (Gan et al. 2020). we are considering extending our fuzzing time, which may obtain better results. Additionally, we only compare CMFuzz with four mainstream fuzzers due to time constraints. We thus plan to compare our work with other state-of-the-art schemes such as FairFuzz (Lemieux and Sen 2018) and Angora (Chen and Chen 2018).

7 Related Work

In this section, we briefly introduce the related work.

Mutation-Based Fuzzing Mutation-based fuzzing becomes popular in security testing especially since AFL (Zalewski 2019) and its descendants (Böhme et al. 2016; Böhme et al. 2017;

Table 6 Number of paths, maximum depth, and crashes found by various fuzzers on LAVA-M

Fuzzers	who			uniq			base64			md5sum		
	Paths	Depth	Crashes	Paths	Depth	Crashes	Paths	Depth	Crashes	Paths	Depth	Crashes
PTfuzz	70	7	2	128	10	0	149	8	0	199	11	6
CMFuzz-PT	207	19	2	161	10	1	197	10	0	356	15	9
Thompson-PT	141	13	0	155	10	0	156	12	0	359	13	6
Greedy-PT	107	16	0	167	10	0	172	10	0	379	11	7
AFL	127	6	1	83	3	2	132	19	0	386	19	8
CMFuzz-AFL	178	19	2	94	14	2	137	20	3	389	24	17
AFLFast	42	7	0	93	7	0	123	9	0	200	12	5
CMFuzz-AFLFast	107	7	0	113	9	0	146	13	0	237	20	5

Gan et al. 2018; Lemieux and Sen 2018) have shown their effectiveness. AFLFast (Böhme et al. 2016) uses a Markov chain model to prioritize path with low-frequency, and CollAFL (Gan et al. 2018) provides more accurate coverage information to mitigate path collisions. Also, FairFuzz (Lemieux and Sen 2018) identifiers and mutates rare branches with lightweight program analysis and heuristics. In addition, AFLGo (Böhme et al. 2017) is a directed greybox fuzzer, which utilizes a simulated annealing approach to optimize power schedule. Another recent work Hawkeye (Chen et al. 2018) is inspired from AFLGo however provides significant improvements on both the static and dynamic analysis. In Hawkeye, the authors collected the information such as the call graph to generate the dynamic metrics that can guide the fuzzer towards the target sites effectively.

In addition, taint analysis can be used to locate the critical byte for guiding the mutation (Rawat et al. 2017; Chen and Chen 2018; Jain et al. 2018). Angora (Chen and Chen 2018) adopts byte-level taint tracking, type inference, and search strategy based on gradient descent to solve path constraints efficiently. VUzzer (Rawat et al. 2017) develops an application-aware evolutionary fuzzing strategy that can infer important input properties via static analysis and dynamic taint analysis. Similarly, TIFF (Jain et al. 2018) performs type-based mutation by using dynamic taint analysis and in-memory data-structure identification. In addition, a recent study, GREYONE (Gan et al. 2020), prioritizes input bytes using taint provided by fuzzing-driven taint inference (FTI).

On a different spectrum, to explore hard-to-reach path, there have been approaches that leverage symbolic execution or concolic execution to generate inputs (Godefroid et al. 2012;

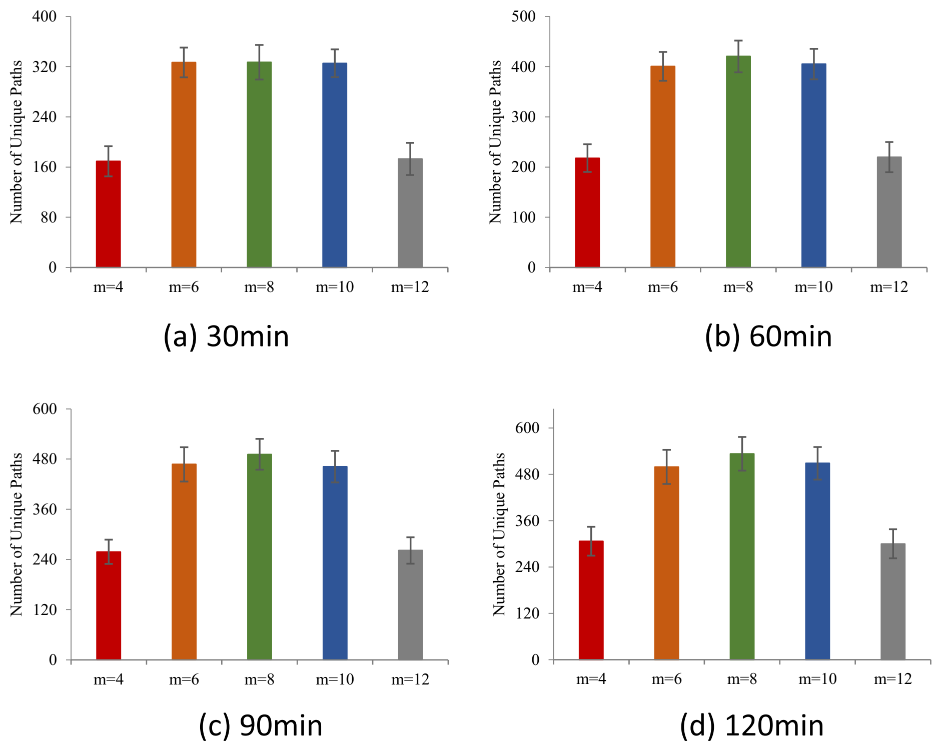


Fig. 7 Number of unique paths discovered by various *m* in 2 h

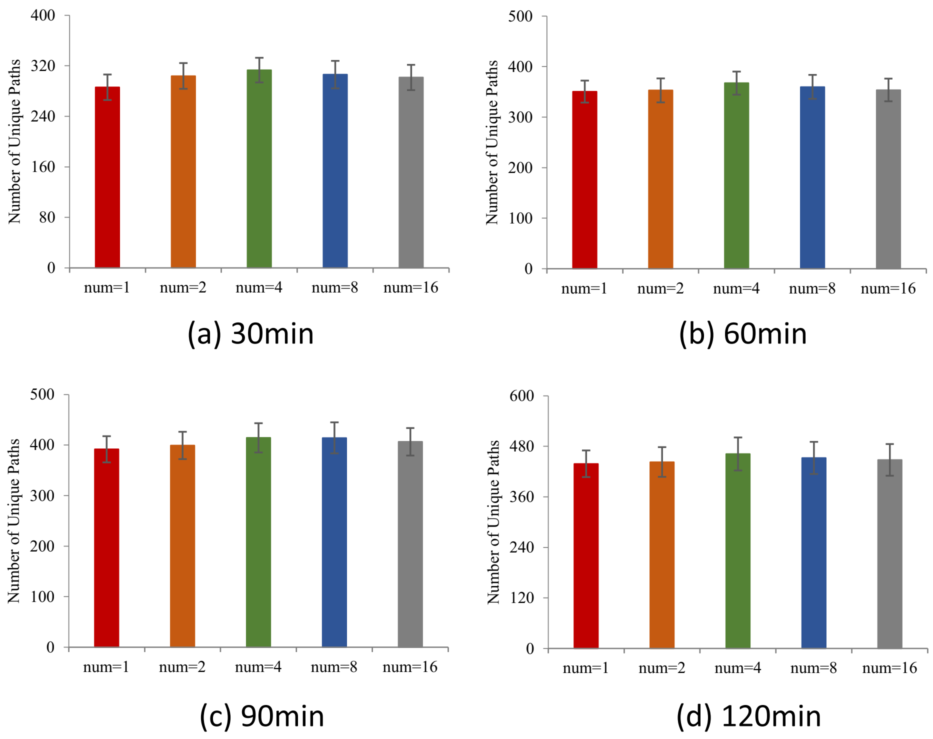


Fig. 8 Number of unique paths discovered by various *num* in 2 h

Stephens et al. 2016; Yun et al. 2018; Zhao et al. 2019). Driller (Stephens et al. 2016) combines fuzzing and concolic execution in a complementary way to generate inputs that can trigger deeper bugs. QSYM (Yun et al. 2018) designs a fast concolic execution engine that integrates symbolic emulation with the native execution to support hybrid fuzzing. More recently, DigFuzz (Zhao et al. 2019) prioritizes and assigns difficult paths for concolic execution via Monte Carlo based probabilistic path prioritization model. Unfortunately, most of these symbolic-based and taint-based solutions cannot scale to large complicated applications due to heavy weight performance overheads.

Our solution CMFuzz utilizes a contextual-bandit algorithm to guide the seed mutation process and generate interesting seeds that can cover more new paths. In addition, CMFuzz can be combined with most of the above-mentioned fuzzers as well as other generation-based fuzzers due to its well-deserved compatibility, such as AFLFast and Skyfire (Wang et al. 2017).

Generation-Based Fuzzing Generation-based fuzzing generates new seeds from a specification (Jibesh Patra 2016; Wang et al. 2017; Godefroid et al. 2017; Lyu et al. 2018). TreeFuzz (Jibesh Patra 2016) learns a generative model of tree structure from a corpus of example data. Skyfire (Wang et al. 2017) presents a data-driven approach to generate well-distributed seeds via leveraging knowledge including syntax features and semantic rules learned from the vast amount of existing samples. Learn&Fuzz (Godefroid et al. 2017) provides a LSTM-based machine learning approach to automatically generate highly-structured well-formed files such as PDF. Similarly, SmartSeed (Lyu et al. 2018) constructs a generative model based on GAN to fast generate valuable binary seed files for fuzzing applications without of requiring highly-structured input format.

Hardware Feedback Aided Fuzzing Several works focused on how to improve the performance of fuzzing using new hardware feature, as proposed in PTfuzz (Zhang et al. 2018) and KAFL (Schumilo et al. 2017). PTfuzz (Zhang et al. 2018) leverages the CPU hardware mechanism Intel Processor Tracer (Intel PT) to collect branch information instead of compile-time instrumentation (AFL) or runtime instrumentation (QAFL). Intel PT (Intel Corporation 2019) can accurately trace program control flow information such as conditional jump and unconditional jump with minimal performance overhead. Therefore, PTfuzz can deal with binaries and achieves smaller overhead than QAFL. Recently, another research KAFL (Schumilo et al. 2017) is also implemented based on Intel PT, which can fuzz OS kernel.

Machine Learning Aided Fuzzing It is worth mentioning that recently machine learning techniques have been extensively applied in improving fuzzing (Rajpal et al. 2017; She et al. 2018). Neuzz (She et al. 2018) guides the fuzzing seed generation process using a deep neural network, while NeuFuzz (Wang et al. 2019b) guides intelligent seed selection using a deep neural network. RETECS (Spieker et al. 2017) employ the reinforcement learning for automatically select and prioritize seed according to their duration, previous last execution, and failure history. Recently, V-Fuzz (Li et al. 2019) uses a neural network-based vulnerability prediction model to estimate which parts of the program may be vulnerable, and then guides vulnerability-oriented evolutionary fuzzer to generate seeds that are closer to the vulnerable locations.

Seed Mutation Strategies Most of the above fuzzers leave the mutation strategies untouched, defaulting to the blind random mutation strategies that comes with AFL, which is inefficient in finding program bugs (Karamcheti et al. 2018; Lyu et al. 2019). To alleviate such a limitation, Böttinger et al. (2018) frames fuzzing as a reinforcement learning problem and utilizes deep Q-learning (Watkins and Dayan 1992) to optimize mutation strategies. Similarity, FuzzerGym (Drozd and Wagner 2018) also uses deep Q-learning to improve mutation selection of libFuzzer (Serebryany 2019). However, these approaches have not shown significant improvements yet according to their experimental results, and their baseline is a random fuzzer or libFuzzer not AFL.

In design closest to our proposal is the work of Karamcheti et al. (2018), which also uses a bandit-based optimization approach (i.e., Thompson Sampling algorithm) to choose adaptively mutation operators. However, as mentioned earlier in this paper, CMFuzz's unique contextual mutation makes it much more powerful than Thompson Sampling. Another recent work MOPT (Lyu et al. 2019) uses a customized particle swarm optimization (PSO) algorithm to select the optimal probability distribution of mutation operators.

While these approaches are quite interesting, they ignore the characteristics of seed files. Here, we primitively take the byte stream as the characteristics of seed files and use them to construct a contextual mutation strategy.

8 Conclusion

This paper presented CMFuzz, a novel context-aware adaptive mutation scheme that utilizes a contextual bandit algorithm, i.e., LinUCB, to guide mutation in fuzzing. We further demonstrated how optimal mutation operators can be effectively selected in each round of fuzzing via contextual information of seed files. To this end, CMFuzz dynamically extracts and encodes the characteristic of seed file as the context to achieve context-aware adaptive mutation strategy. We implemented the proposed mutation scheme in several mutation-based fuzzers such as PTfuzz, AFL, and AFLFast. Our evaluation showed that CMFuzz can significantly outperform other state-of-the-art mutation-based fuzzers such as PTfuzz in discovering unique paths, unique crashes, and maximum depth on most cases. More importantly, we also conducted model evaluation to demonstrate that LinUCB exceeded other bandit algorithms such as Thompson Sample and epsilon-greedy.

Acknowledgments We would like to thank all practitioners who participated in the focus group discussions, as well as we thank the anonymous reviewers for their constructive comments to improve the paper. This work was supported by the National Key R&D Program of China under Grant 2016QY07X1404.

References

Adobe (2019) A Basic Distributed Fuzzing Framework for FOE.https://blogs.adobe.com/security/2012/05/a-basic-distributed-fuzzing-framework-for-foe. html. Accessed 5 Dec 2019

Agrawal S, Goyal N (2011) Analysis of Thompson sampling for the multi-armed bandit problem. J Mach Learn Res 23:1–26

Aschermann C, Schumilo S, Blazytko T et al (2019) REDQUEEN: fuzzing with input-to-state correspondence. In: Proceedings of the 2019 network and distributed system security symposium. The Internet Society, San Diego, pp 1–15

Böhme M, Pham V-T, Roychoudhury A (2016) Coverage-based Greybox Fuzzing as Markov Chain. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16. ACM Press, New York, pp. 1032–1043

Böhme M, Pham V-T, Nguyen M-D, Roychoudhury A (2017) Directed Greybox Fuzzing. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17. ACM Press, New York, pp. 2329–2344

Böttinger K, Godefroid P, Singh R (2018) Deep Reinforcement Fuzzing. In: Proceedings of the 2018 IEEE Security and Privacy Workshops (SPW). IEEE, San Francisco, pp. 116–122

Cha SK, Woo M, Brumley D (2015) Program-adaptive mutational fuzzing. In: Proceedings of the 2015 IEEE Symposium on Security and Privacy. IEEE, San Jose, pp. 725–741

Chen P, Chen H (2018) Angora: efficient fuzzing by principled search. In: Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP). IEEE, San Francisco, pp 711–725

Chen H, Xue Y, Li Y, et al (2018) Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM, New York, pp. 2095–2108

Dhillon IS, Sra S (2005) Generalized Nonnegative Matrix Approximations with Bregman Divergences. Advances in Neural Information Processing Systems 283–290

Dolan-Gavitt B, Hulin P, Kirda E, et al (2016) LAVA: large-scale automated vulnerability addition. In: Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP). IEEE, San Jose, pp. 110–121

Drozd W, Wagner MD (2018) FuzzerGym: a competitive framework for fuzzing and learning. arXiv e-prints

Gan S, Zhang C, Qin X, et al (2018) CollAFL: path sensitive fuzzing. In: Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP). IEEE, San Francisco, pp. 679–696

Gan S, Zhang C, Chen P, et al (2020) GREYONE: Data Flow Sensitive Fuzzing. Proceedings of the 2020 USENIX Security Symposium. USENIX, Boston, pp. 1–18

Godefroid P, Levin MY, Molnar D (2012) SAGE: Whitebox fuzzing for security testing. Commun ACM 55:40–44. https://doi.org/10.1145/2093548.2093564

Godefroid P, Peleg H, Singh R (2017) Learn&Fuzz: machine learning for input fuzzing. In: Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, Urbana-Champaign, pp. 50–59

Google (2019a) ClusterFuzz. https://google.github.io/clusterfuzz/.

Google (2019b) OSS-Fuzz. https://google.github.io/oss-fuzz/.

Han H, Cha SK (2017) IMF: Inferred Model-based Fuzzer. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17. ACM Press, New York, pp. 2345–2358

Intel Corporation (2019) Intel Processor Trace. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing. Accessed 5 Dec 2019

Jain V, Rawat S, Giuffrida C, Bos H (2018) TIFF: using input type inference to improve fuzzing. In: Proceedings of the 34th Annual Computer Security Applications Conference. ACM, New York, pp. 505–517

Jibesh Patra MP (2016) Learning to fuzz: application-independent fuzz testing with probabilistic, generative models of input data. TU Darmstadt, Department of Computer Science 1:123–129

Karamcheti S, Mann G, Rosenberg D (2018) Adaptive Grey-box fuzz-testing with Thompson sampling. In: Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security - AISec '18. ACM Press, New York, pp. 37–47

Lee DD, Seung HS (1999) Learning the parts of objects by nonnegative matrix factorization. Nature 401(6755): 788–791

Lee DD, Seung HS (2001) Algorithm for non-negative matrix factorization. Adv Neural Inf Proces Syst:556–562

Lemieux C, Sen K (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 - ASE 2018. ACM Press, New York, pp. 475–485

Lemieux C, Padhye R, Sen K, Song D (2018) PerfFuzz: Automatically Generating Pathological Inputs. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2018. ACM Press, New York, pp. 254–265

Li L, Chu W, Langford J, Schapire RE (2010) A contextual-bandit approach to personalized news article recommendation. In: Proceedings of the 19th international conference on World wide web - WWW '10. ACM Press, New York, pp. 1–10

Li Y, Chen B, Chandramohan M, et al (2017) Steelix: Program-state based Binary Fuzzing. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017. ACM Press, New York, pp. 627–637

Li Y, Ji S, Lv C, et al (2019) V-fuzz: vulnerability-oriented evolutionary fuzzing. arXiv e-prints 1–16

Libpng (2019) http://www.libpng.org/pub/png/libpng.html. Accessed 13 Dec 2019

Lyu C, Ji S, Li Y, et al (2018) SmartSeed: smart seed generation for efficient fuzzing. arXiv e-prints 1–17

Lyu C, Ji S, Zhang C, et al (2019) MOPT: optimized mutation scheduling for Fuzzers. In: Proceedings of the 2019 USENIX Security Symposium. USENIX, Santa Clara, pp. 1–21

McCaffrey J (2019) The Epsilon-Greedy Algorithm. https://jamesmccaffrey.wordpress.com/2017/11/30/the-epsilon-greedy-algorithm/. Accessed 4 Oct 2019

Peng H, Shoshitaishvili Y, Payer M (2018) T-fuzz: fuzzing by program transformation. In: Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP). IEEE, San Francisco, CA, USA, pp. 697–710

Rajpal M, Blum W, Singh R (2017) Not all bytes are equal: neural byte sieve for fuzzing. arXiv e-prints 1–10

Rawat S, Jain V, Kumar A, et al (2017) VUzzer: application-aware evolutionary fuzzing. In: Proceedings 2017 Network and Distributed System Security Symposium. Internet Society, Reston, VA, pp. 1–13

Rebert A, Cha SK, Avgerinos T, et al (2014) Optimizing seed selection for fuzzing. In: Proceedings of the 2014 USENIX Security Symposium. USENIX, San Diego, pp. 861–875

Schumilo S, Aschermann C, Gawlik R, et al (2017) kAFL: hardware-assisted feedback fuzzing for OS kernels. In: Proceedings of the 2017 USENIX Security Symposium. USENIX, Vancouver, pp. 167–182

Serebryany K (2019) LibFuzzer. http://llvm.org/docs/LibFuzzer.html. Accessed 4 Dec 2019

She D, Pei K, Epstein D, et al (2018) NEUZZ: Efficient Fuzzing with Neural Program Smoothing. arXiv e-prints

Slivkins A (2019) Introduction to multi-armed bandits. Found Trends® Mach learn 12:1–286. https://doi.org/10.1561/2200000068

Spieker H, Gotlieb A, Marijan D, Mossige M (2017) Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2017. ACM Press, New York, pp. 12–22

Stephens N, Grosen J, Salls C, et al (2016) Driller: augmenting fuzzing through selective symbolic execution. In: Proceedings 2016 Network and Distributed System Security Symposium. Internet Society, Reston, VA, pp. 21–24

Wang J, Chen B, Wei L, Liu Y (2017) Skyfire: data-driven seed generation for fuzzing. In: Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP). IEEE, San Jose, pp. 579–594

Wang J, Duan Y, Song W, et al (2019a) Be sensitive and collaborative: analyzing impact of coverage metrics in Greybox fuzzing. In: Proceedings of the 2019 International Symposium on Research in Attacks, Intrusions and Defenses. USENIX, Beijing, pp. 1–15

Wang Y, Wu Z, Wei Q, Wang Q (2019b) NeuFuzz: Efficient Fuzzing with Deep Neural Network. IEEE Access 7:36340–36352. https://doi.org/10.1109/ACCESS.2019.2903291

Watkins CJCH, Dayan P (1992) Technical note: Q-learning. Mach Learn 8:279–292. https://doi.org/10.1023/A:1022676722315

Woo M, Cha SK, Gottlieb S, Brumley D (2013) Scheduling Black-box Mutational Fuzzing. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13. ACM Press, New York, pp. 511–522

Xu W, Kashyap S, Min C, Kim T (2017) Designing New Operating Primitives to Improve Fuzzing Performance. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17. ACM Press, New York, pp. 2313–2328

You W, Zong P, Chen K, et al (2017) SemFuzz: Semantics-based Automatic Generation of Proof-of-Concept Exploits. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17. ACM Press, New York, pp. 2139–2154

You W, Wang X, Ma S, et al (2019) ProFuzzer: on-the-fly input type probing for better zero-day vulnerability discovery. In: Proceedings of the 2019 IEEE Symposium Security Privacy, IEEE, San Francisco, pp. 1–18

Yun I, Lee S, Xu M, et al (2018) QSYM: practical Concolic execution engine tailored for hybrid fuzzing. In: Proceedings of the 2018 USENIX Security Symposium. USENIX, Baltimore, pp. 745–761

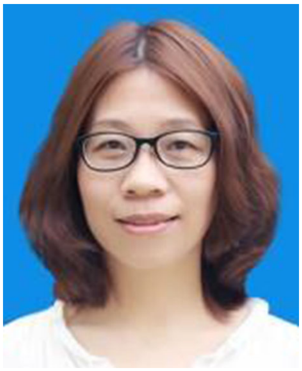
Zalewski M (2019) American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/. Accessed 5 April 2019

Zhang G, Zhou X, Luo Y et al (2018) PTfuzz: guided fuzzing with processor trace feedback. IEEE Access 6: 37302–37313. https://doi.org/10.1109/ACCESS.2018.2851237

Zhao L, Duan Y, Yin H, Xuan J (2019) Send hardest problems my way: probabilistic path prioritization for hybrid fuzzing. In: Proceedings 2019 Network and Distributed System Security Symposium. The Internet Society, San Diego, pp. 1–15

Xiajing Wang received the B.E. degree in software engineering from the Taiyuan University of Technology, China, in 2016. She is currently pursuing the Ph.D. degree with the School of Computer Science and Technology, Beijing Institute of Technology, China. Her research interests include information security and vulnerability discovery.

Changzhen Hu received the Ph.D. degree from Beijing Institute of Technology, China, in 1996. He is a professor with the School of Computer Science and Technology, Beijing Institute of Technology, China. His current research interests include software security, network security, network attack and defense, and vulnerability analysis.



Rui Ma received the Ph.D. degree from Beijing Institute of Technology, China, in 2004. She is an associate professor with the School of Computer Science and Technology, Beijing Institute of Technology, China. Her current research interests include software security, Internet of things, neural network, and data mining.

Donghai Tian received the Ph.D. degree from Beijing Institute of Technology, China, in 2012. He is an associate professor with the School of Computer Science and Technology, Beijing Institute of Technology, China. His current research interests include software security, malware analysis and detection, Android security, and cloud security.

Jinyuan He received the B.E. degree in software engineering from the Beijing Institute of Technology, China, in 2019. She is currently pursuing the M.S. degree with the School of Computer Science and Technology, Beijing Institute of Technology, China. Her research interests include software security and vulnerability analysis.

Affiliations

Xiajing Wang[1] · Changzhen Hu[1] · Rui Ma[1] · Donghai Tian[1] · Jinyuan He[1]

Xiajing Wang
xiajing.wang@foxmail.com

Changzhen Hu
chzhoo@bit.edu.cn

Donghai Tian
donghaitad@gmail.com

Jinyuan He
jyuan.h@qq.com

[1] Beijing Key Laboratory of Software Security Engineering Technology, School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China