# AutoFuzz: Automatic Fuzzer-Sanitizer Scheduling with Multi-Armed Bandit

Yijia Gao[1], Wenrui Zeng[1], Siyuan Liu[1], Yingpei Zeng[1*]

[1]School of Cyberspace, Hangzhou Dianzi University, Hangzhou, 310018, Zhejiang, China.

*Corresponding author(s). E-mail(s): yzeng@hdu.edu.cn;

**Abstract**

Coverage-guided fuzzing (CGF) is a widely used technique for exposing vulnerabilities in software. Despite its success, selecting the ideal CGF fuzzer or fuzzer combination for a particular program continues to pose a challenge, given that no single fuzzer or fuzzer combination consistently outperforms others. Furthermore, the integration of sanitizers during fuzzing also needs to be carefully considered, due to the overhead sanitizers introduced. This paper proposes AutoFuzz, a Multi-Armed Bandit (MAB)-based method that automatically schedules different fuzzing methods (a fuzzer and sanitizer combination) without the need for extensive pre-fuzzing experiments. AutoFuzz utilizes a non-stochastic multi-armed bandit to model the scheduling problem, and employs the Exp3 algorithm to run the MAB. AutoFuzz considers both coverage and crash gains in its reward calculation. Experimental results demonstrate that AutoFuzz performs better than standalone fuzzers and sanitizers and a Round Robin method (i.e., similar scheduling method to EnFuzz and Cupid), presenting a promising solution for efficient fuzzer selection and sanitizer usage during fuzz testing.

**Keywords:** Coverage-guided fuzzing, Fuzzer scheduling, Sanitizer, Multi-armed bandit

## 1 Introduction

Fuzz testing technique especially coverage-guided fuzzing (CGF) (Zalewski, 2017, Böhme et al., 2016) is one of the most popular software testing techniques to expose vulnerabilities in computer programs (Miller et al., 1990, Manes et al., 2019, Zhu et al., 2022, Mallissery and Wu, 2023). This is because CGF fuzzing could gradually explore the state space of the program under test (PUT) by a fuzzing loop: mutating seeds to create new inputs, tracing the coverage information of the new inputs, and adding the inputs having new code coverage into the seed pool as new seeds. The tracing of coverage information is achieved by instrumenting the PUT before the fuzzing, and the PUT may be further instrumented with sanitizers like AddressSanitizer (ASan) (Serebryany et al., 2012) for better uncovering vulnerabilities. The famous OSS-Fuzz project (Google Security Team, 2018) using CGF fuzzers like AFL (American Fuzzy Lop) (Zalewski, 2017), libFuzzer (LLVM, 2023), AFL++ (Fioraldi et al., 2020), and honggfuzz (Google, 2023), has discovered over 10,000 vulnerabilities and 36,000 bugs across 1,000 open-source projects by August 2023 [1]. CGF fuzzers have also been used to fuzz operating systems (Google, 2015, Pan et al., 2021, Liu et al., 2023), network protocols (Pham et al., 2020, Andronidis and Cadar, 2022, Ba et al., 2022), databases (Liang et al., 2022, Jiang et al., 2023, Zeng et al., 2023), distributed systems (Meng et al., 2023, Chen, 2024), and Internet of Things (IoT) (Zheng et al., 2019, Zeng et al., 2020, Zhu et al., 2023).

For a given particular PUT, choosing the ideal CGF fuzzer for it is not easy, since currently, no fuzzer performs the best for all programs (Li et al., 2021, Hazimeh et al., 2020, Metzman et al., 2021, Liu et al., 2023). One possible method is to try all fuzzers first with experiments. But even if we try all fuzzers for a PUT and find the ideal fuzzer for it, when we want to reuse the finding for fuzzing new versions of the PUT (e.g., continuous fuzzing (Google Security Team, 2018)) the ideal fuzzer may change when new versions of the PUT or the fuzzer are released. In addition, fuzzers may complement each other (Chen et al., 2019), which means the combination of some fuzzers may outperform a single type of fuzzers (Chen et al., 2019). Furthermore, fuzzers may also be combined with different proportions (e.g., 1:2, one fuzzer A instance and two fuzzer B instances). As a result, finding out the best combination of fuzzers for a PUT may need too many experiments, and may outweigh the efficiency gain brought by the combination. Consequently, in practice, users usually choose a relatively good fuzzer (Metzman et al., 2021) or a relatively good combination of fuzzers (Chen et al., 2019, Güler et al., 2020) to fuzz the PUT.

Additionally, whether to enable sanitizers and which sanitizers to enable have not been researched well in the literature. Sanitizers make vulnerabilities more apparent to expose by crashing the program immediately when the vulnerabilities happen. Some users choose to use sanitizers during fuzzing, e.g., running one fuzzer instance for each kind of sanitizer (Google Security Team, 2018). However, sanitizers also bring considerable overhead, for example, AddressSanitizer (ASan) helps the detection of vulnerabilities including heap buffer overflow, but incurs about 2x the slowdown of the program execution (Serebryany et al., 2012). If the PUT does not have the vulnerabilities the used sanitizer is targeting, the fuzzing is unnecessarily slowed down (Song et al., 2019). Consequently, some users opt to exclude sanitizers during the fuzzing process, preferring instead to re-run the seeds after fuzzing to identify vulnerabilities or eliminate duplicates (Manes et al., 2019, Lyu et al., 2019, Jauernig et al., 2023). However, this approach may result in missed vulnerabilities that could have been detected during fuzzing.

[1] https://github.com/google/oss-fuzz

In this paper, we propose a method named AutoFuzz to automatically schedule different fuzzing methods (i.e., fuzzer instances with sanitizers) for higher performance based on the Multi-Armed Bandit (MAB) scheme, and with AutoFuzz, there is no need to use pre-fuzzing experiments to find out the ideal fuzzers, ideal sanitizers, or the ideal proportion of them. AutoFuzz models the different choices (e.g., fuzzing methods) as different arms of a non-stochastic multi-armed bandit, and uses the famous MAB algorithm Exp3 (Auer et al., 2002) to select the optimal arms and balance between exploration and exploitation (Section 3.5). For determining the reward of each MAB arm selection in the Exp3 algorithm, AutoFuzz uses a new reward calculation method (Section 3.4) to consider both the crash (Section 3.2) and coverage (Section 3.3) gains. We implement a prototype of AutoFuzz that supports the state-of-the-art fuzzers like AFL++, libFuzzer, MOpt (Lyu et al., 2019), and sanitizers include AddressSanitizer (ASan), UndefinedBehaviorSanitizer (UBSan) (LLVM, 2023), and use it to fuzz 13 open-source programs. The results show that AutoFuzz could automatically schedule the proper fuzzers and sanitizers and get a higher reward than both standalone fuzzers or sanitizers, and a Round Robin method (similar to the EnFuzz (Chen et al., 2019) and Cupid (Güler et al., 2020) scheduling methods).

The main contributions of our study are as follows:

- We propose AutoFuzz to dynamically schedule different fuzzing methods at runtime for better vulnerability discovery with existing fuzzers and sanitizers.
- We implement a prototype of AutoFuzz and compare it with the state-of-the-art fuzzers and sanitizers and a Round Robin method (i.e., the EnFuzz&Cupid scheduling), and show it has better performance. We open source it at https://github.com/AutoFuzz/AutoFuzz for research usage.

This paper is organized as follows. In Section 2, we introduce the background and the motivation behind our research. Section 3 provides a detailed exposition of the proposed AutoFuzz approach. In Section 4 we present and discuss the experimental results obtained. In Section 5 we discuss the limitations of our evaluation and AutoFuzz. Section 6 is dedicated to an exploration of related works in the field. Finally, we draw conclusions in Section 7.

## 2 Background and Motivation

### 2.1 Coverage-guided Fuzzing (CGF)

We briefly introduce coverage-guided Fuzzing here by using AFL (Zalewski, 2017), a CGF fuzzer, holds widespread adoption in both academia and industry, influencing the development of fuzzers like AFL++ (Fioraldi et al., 2020), libFuzzer (LLVM, 2023) and others (Li et al., 2021, Lyu et al., 2019, Lemieux and Sen, 2018, Yue et al., 2020) in the literature. Key modules of AFL include coverage tracing and seed mutation.

AFL relies on lightweight instrumentation for coverage tracing, which will record execution paths in a bitmap. Each edge of an execution path is mapped to a random byte, which utilizes an 8-bit counter to record execution counts. Inputs achieving new coverage, determined by new edges or counter buckets, are added to the seed pool and maintained as a queue.

Seed mutation involves iterating through the seed queue, selecting seeds probabilistically based on factors like fuzzing history and favoring marked seeds. A deterministic stage may precede fuzzing for previously untouched seeds. The indeterministic stage, comprising havoc and splicing stages, involves mutating seeds based on a score incorporating execution time, coverage, and discovery time. Havoc stage mutations are performed randomly, influenced by a mutation count parameter. A dictionary containing keywords may be used for replacing contents in seeds during mutation. The splicing stage involves randomly splicing seeds with others first, and after that, it is similar to the havoc stage.

## 2.2 Sanitizer

Sanitizers (Serebryany et al., 2012, Song et al., 2019) are becoming crucial tools in software development for identifying and mitigating various types of vulnerabilities (Song et al., 2019). AddressSanitizer (ASan) (Serebryany et al., 2012) is the mostly used sanitizer (Song et al., 2019). It focuses on memory safety issues, particularly detecting memory corruption errors such as buffer overflows and use-after-free bugs. ASan achieves this by instrumenting the code during compilation and adding runtime checks to detect invalid memory access. In the case of ASan, the sanitizer maintains a shadow memory region to track the state of each byte in the program's memory. When the program executes, ASan checks this shadow memory to detect out-of-bounds accesses. Upon identifying an issue, ASan crashes the program and provides detailed error reports, including information about the type and location of the error, facilitating efficient debugging. ASan has an average 2x slowdown to the execution due to the instrumentation it adds to the program (Serebryany et al., 2012).

Other sanitizers are less used due to compatibility issues (Song et al., 2019) but are useful for detecting specific vulnerabilities. For example, UndefinedBehaviorSanitizer (UBSan) (LLVM, 2023) is another sanitizer that targets undefined behaviors in programs. UBSan assists in detecting issues like integer overflows, misaligned memory access, and other undefined behaviors specified by the language standards. Similar to ASan, UBSan instruments the code to insert runtime checks, flagging violations during program execution.

People may use sanitizers during fuzzing to guarantee crashes when vulnerabilities happen (Google Security Team, 2018), or instead of using sanitizers during fuzzing, they may just use extra PUTs with sanitizers enabled to re-run the seeds after fuzzing to discover vulnerabilities and remove duplicated vulnerabilities (Manes et al., 2019, Lyu et al., 2019, Jauernig et al., 2023), which could avoid the execution slowdown brought by sanitizers.

## 2.3 Motivation

When a user wants to fuzz a program to find out its vulnerabilities, she/he needs to first determine which CGF fuzzer or fuzzer combination to use. In addition, she/he should determine whether to enable sanitizers and which sanitizers to enable during fuzzing. However, currently, it is difficult for her/him to decide due to the following problems.

- (I) Currently, no CGF fuzzer or fuzzer combination constantly outperforms other CGF fuzzers or fuzzer combinations when fuzzing all programs (Li et al., 2021, Metzman et al., 2021, Chen et al., 2019). For example, MOpt (Lyu et al., 2019) is an innovative fuzzer that tries to find the optimal selection probability distribution of mutation operators and has also been integrated into AFL++ (Fioraldi et al., 2020) in an optional mode. However, MOpt finds the most vulnerabilities in programs like pdftotext and imginfo (Li et al., 2021), but has a relatively low code coverage in other programs like freetype2-2017 (Metzman et al., 2021).
- (II) The performance of a fuzzer may change with its added features and new versions of the same PUT. For example, AFL experiences a performance boost for certain programs when the FidgetyAFL feature is incorporated in v2.31b (Zalewski, 2017, Böhme et al., 2016). In the case of EnFuzz, its different fuzzer combinations are ranked differently in terms of branch coverage results for openssl-1.0.1 and openssl-1.0.2 (Chen et al., 2019).
- (III) The performance of a fuzzer may vary with different fuzzing settings, e.g., the initial seed corpus and dictionary, which may undergo changes during program development. Klees et al. have demonstrated that different numbers of seeds have a great impact on the performance of CGF fuzzers (Klees et al., 2018). In (Metzman et al., 2021) it is also shown that employing different seed corpora leads to substantial changes in the ranking of fuzzers, while the use or non-use of a dictionary has only a minor impact.
- (IV) Enabling sanitizers during fuzzing may not be worthwhile. We use the code snippet shown in Listing 1 as an example. It contains two stack-buffer-overflow vulnerabilities. The overflow in `func2` usually can be detected even ASan is not enabled during fuzzing, since the overflow will crash the program when `len` has a large value such as 2048. However, the overflow in `func1` can only be detected when ASan is enabled during fuzzing. This is because the overflow merely overwrites a neighboring unimportant byte in the stack and does not lead to a program crash (tested with LLVM-13 and Ubuntu 22.04). Moreover, the overflow occurs only when `len==5`, and usually, corresponding seeds are not saved (no new coverage). Consequently, re-running the seeds with PUT and enabling sanitizers after fuzzing will not induce a crash either. If the PUT does not encompass vulnerabilities that can solely be detected with a sanitizer enabled, it might not be worthwhile to enable the sanitizer because of the execution slowdown it entails.

Current solutions exhibit their limitations. One solution involves experimenting with all fuzzers and sanitizers for the program to determine the optimal fuzzer or fuzzer combination (Li et al., 2021, Metzman et al., 2021, Chen et al., 2019, Güler et al., 2020) and whether to utilize sanitizers. The solution attempts to address problems I and IV. However, firstly, this approach can be extremely time-consuming, as it requires experimenting with all fuzzers and sanitizers prior to initiating the actual fuzzing campaign. For instance, each fuzzer and sanitizer may need to be tested for 12 or more hours and repeated at least five times (Klees et al., 2018, Böhme et al., 2022). The time commitment increases even further when considering combinations of multiple fuzzers. Thus, the solution is only beneficial when the program is to be fuzzed many times (e.g., integrating fuzzing into the software development process like the OSS-Fuzz

```c
void func2(int len)
{
   char *str[10];
   str[len]='a'; // an overflow that may crash without ASan,
                 // i.e., when len has a big value
 }
void func1(int len)
{
   int a[5];
   func2(len);
   if (len<=5)
       a[len]=1;  // an overflow that only crashes with ASan
                  // when len==5
 }
```

Listing 1: A code snippet for the vulnerability detection with Address Sanitizer (ASan).

project (Google Security Team, 2018) that keeps fuzzing new development versions of the open-source projects). Secondly, this solution cannot overcome problems II and III since both fuzzers and the program are constantly evolving, and the fuzzing settings may change, such as the addition of new initial seeds. Another solution is to adopt an ad hoc approach in fuzzer and sanitizer selection, which entails the risk of not employing the best option (i.e., choosing a random solution for problems I, II, III, and IV). A user with numerous CPU cores may utilize multiple good fuzzers that rank highly in the benchmarks simultaneously, allocate equal CPU resources to each of them, and also enable all sanitizers in the PUT, as done in the OSS-Fuzz project (Google Security Team, 2018). A user with a limited number of CPU cores may use only one good fuzzer and refrain from enabling sanitizers during fuzzing but may use them to re-run seeds after fuzzing.

In contrast, AutoFuzz addresses all of the aforementioned problems I, II, III, and IV without requiring prior experimentation with fuzzers and sanitizers. It accomplishes this by automatically learning the optimal fuzzers and sanitizers for the PUT during the fuzzing process.

## 3 Approach

We will give an overview of our approach first and explain its important components in subsections.

### 3.1 Overview

We propose avoiding preliminary fuzzing experiments to assess the performance of fuzzers, sanitizers, or fuzzer combinations with a given program. Such preliminary evaluations typically require enough fuzzing time to gauge a fuzzer's performance accurately. This approach is useless for users aiming to fuzz a program only once, as the time invested in preliminary learning offers little benefit. Even for users who repeatedly fuzz a program (e.g., for secure software development like OSS-Fuzz (Google Security Team, 2018)), this method is inefficient since the information rapidly becomes outdated as fuzzers and programs evolve, as discussed in Section 2.3.

Instead, we propose an approach that learns the performance of a fuzzer and a sanitizer with a program in real-time during the fuzzing campaign, using this information to guide the scheduling of fuzzers and sanitizers concurrently. This real-time learning ensures the accuracy of the information. Reinforcement learning is particularly suitable for this task. In general reinforcement learning, an agent interacts with its environment in discrete time steps (rounds). The agent takes an action each round, receives a corresponding reward, and transitions to a new state. The agent uses the reward feedback to adjust its future actions. Therefore, it is crucial to model our problem appropriately as a reinforcement learning problem.

In this study, we introduce a novel approach called AutoFuzz, which leverages reinforcement learning to automate the scheduling of various fuzzing methods (i.e., fuzzer instances, detailed subsequently). We divide a fuzzing campaign into uniform time slots, each corresponding to a discrete step within a reinforcement learning framework. In this setup, a scheduler acts as the agent, whose action is to select a fuzzing method for each round. To address the dynamic selection of fuzzing methods, we frame the problem as a multi-armed bandit (MAB) problem (Auer et al., 2002), a well-established problem in classic reinforcement learning that has been employed in several fuzzing studies (Yue et al., 2020, Wang et al., 2021, Wu et al., 2022, Zhang et al., 2022, Lee et al., 2023, Lin et al., 2024) due to its simplicity and efficiency. In this context, the selection of fuzzing methods is mapped to choosing an arm of the bandit, with rewards defined by the coverage and crashes achieved. Utilizing an MAB algorithm allows us to maximize the total reward within a finite number of rounds by strategically selecting the optimal arms.

The architecture of AutoFuzz is shown in Figure 1. Its main modules are briefly explained as follows.

- Fuzzing Method: A fuzzing method is a fuzzer instance determined by both its fuzzer and its PUT (including any sanitizers if used). For instance, the *AFL++ ASan* fuzzing method represents an AFL++ fuzzer paired with a PUT instrumented with ASan, while the *AFL++ Pure* fuzzing method represents an AFL++ fuzzer paired with an ordinary PUT (no sanitizers enabled). Selecting a fuzzing method entails choosing both its fuzzer and sanitizer.
- Reward Calculation (Section 3.4): The reward calculation module is designed to compute the reward for each round execution of a fuzzing method. It relies on crash analysis (Section 3.2) and coverage analysis (Section 3.3), two modules dedicated to analyzing the crashes and coverage obtained during fuzzing.
- Scheduler (Section 3.5): The scheduler is responsible for selecting different fuzzing methods in each time slot (round), and it uses a multi-armed bandit algorithm Exp3 (Auer et al., 2002) to determine the selection. The scheduler will also synchronize the seed corpus of the chosen fuzzing method in each round to the latest seed corpus before starting the fuzzing method. It also invokes the reward calculation module to obtain rewards after completing the selected fuzzing methods.

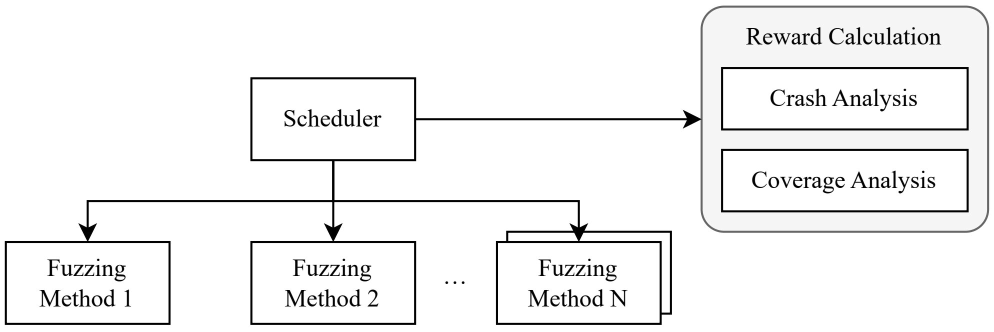


**Fig. 1**: The overall architecture of AutoFuzz

## 3.2 Crash Analysis

The crash analysis module offers a robust and fuzzer-agnostic approach to identifying unique crashes. Various existing fuzzers may categorize a crash as unique under varying conditions, which might not conform to a standardized criterion. For instance, AFL considers a crash as unique if it possesses a distinct execution bitmap (Zalewski, 2017), whereas a more widely accepted method relies on stack hashing (Manes et al., 2019, Klees et al., 2018). In our quest to automatically and impartially analyze unique crashes, taking into consideration both their prevalence and feasibility, we have opted to adopt the stack hashing-based method, utilizing the most recent three stack frames to generate the hash (Manes et al., 2019, Li et al., 2021, Klees et al., 2018). For the sake of simplicity, we will interchangeably use the terms "crash" and "vulnerability" throughout this paper.

The module's algorithm is shown in Algorithm 1. This algorithm takes as input the newly collected crash files (FileList) and previously collected crashes (UniqueCrashes, null if called for the first time). To enhance efficiency, we employ multiple threads for concurrent analysis. Consequently, crashes are partitioned into groups and distributed to individual analysis threads for parallel processing. For each crash, we attempt rerunning the crash file using a test program. Initially, the program instrumented by ASan serves as the test program, and if no crashes are detected, we resort to the program that initially uncovered the vulnerability. Subsequently, the test program generates the corresponding output result containing call stack information. We then compute the hashes of the crash stack and XOR the top three recent ones to derive a single hash value, which serves as the unique identifier for the crash. By comparing this hash with those in UniqueCrashes, we can determine whether the vulnerability has occurred or not and increment the corresponding counter accordingly. Both new crashes and duplicated crashes have counters, as they are utilized in the reward calculation module explained later in Section 3.4.

## 3.3 Coverage Analysis

The coverage analysis module is dedicated to assessing the code coverage achieved by a fuzzing method within a round. Although our ultimate objective is to discover

**Algorithm 1** Crash Analysis

```
Input: FileList (The crash file list), UniqueCrashes (Existing unique crashes)
Output: NewCrashNum (Number of new crashes), DupCrashNum (Number of duplicated
    crashes)
1:  RunResults ← $\phi$
2:  NewCrashNum ← $\phi$, DupCrashNum ← $\phi$
3:  Groups ← DivideIntoGroups(FileList)              ▷ Divide into multiple groups
4:  RunResults ← RunFilesInParallel(Groups)     ▷ Run files as inputs in background
5:  for all Result∈ RunResults do
6:      Hashes ← GetHashStacks(Result)
7:      Hash ← XorTop3Stacks(Hashes)
8:      if Hash $\notin$ UniqueCrashes then
9:          UniqueCrashes.Add(Hash)
10:         NewCrashNum[Result.VulnerType] ← NewCrashNum[Result.VulnerType] + 1
11:     else
12:         DupCrashNum[Result.VulnerType] ← DupCrashNum[Result.VulnerType] + 1
13:     end if
14: end for
```

vulnerabilities in the PUT, the fuzzing methods must first discover the code regions potentially harboring vulnerabilities (Klees et al., 2018, Böhme et al., 2022). Moreover, uncovering vulnerabilities in the early stages of testing can be challenging for fuzzing methods (since vulnerabilities are usually rare); thus, code coverage serves as a primary metric for these methods to attain rewards. The gained coverage is measured by calculating the code coverage of all newly discovered seeds of the fuzzing method during the round.

The LLVM toolset provides llvm-cov and llvm-profdata, facilitating the acquisition of code coverage for the fuzzing methods. Leveraging these tools, we can obtain coverage metrics such as branch, function, instantiation, line, and region. Consistent with prevalent practices in existing research (Manes et al., 2019, Zeng et al., 2023, Li et al., 2021, Chen et al., 2019, Jauernig et al., 2023, Klees et al., 2018, Lin et al., 2022, Chen et al., 2023), we opt for branch (i.e., edge) coverage as our metric. The implementation of the coverage analysis module closely resembles that of the crash analysis module. However, instead of crash files, seed files are provided as input to the module, and the processing is on measuring code coverage rather than the number of crashes.

## 3.4 Reward Calculation

The purpose of the reward calculation module is to compute the reward for a fuzzing method after it has been scheduled once. This module takes the outputs of the crash analysis and coverage analysis modules as inputs and generates a numerical value representing the reward. AutoFuzz leverages not only discovered crashes but also increased coverage to calculate rewards, as crash discovery is sparse, whereas denser rewards help the learning algorithm converge (Ng et al., 1999). Recent research also demonstrates a strong correlation and moderate agreement between coverage achieved and bugs found (Klees et al., 2018, Böhme et al., 2022).

Firstly, we denote $R_{crash}$ as the reward corresponding to discovered crashes. This reward is designed based on two principles. First, different types of vulnerabilities may carry different weights in the reward, with $W_i$ representing the weight assigned to type $i$ vulnerabilities. Second, newly discovered vulnerabilities are deemed most significant and thus receive a standard weight, while duplicated vulnerabilities are assigned a lower weight, though not zero. This approach aims to incentivize fuzzing methods to discover vulnerabilities more frequently (Ng et al., 1999).

$$R_{crash} = \sum_{i \in type} (\text{NewCrashNum[i]} + \text{DupCrashNum[i]}W_{dup})W_i \tag{1}$$

Secondly, we employ $R_{cov}$ to denote the reward for code coverage, which is computed straightforwardly. We calculate the increase in code coverage for the current round and divide it by the total coverage, multiplying by a weight $W_{br}$ to obtain the reward.

$$R_{cov} = \frac{\text{coverage}_{inc}}{\text{coverage}_{total}} W_{br} \tag{2}$$

Finally, AutoFuzz normalizes the two rewards and obtains the final reward $R$. Specifically, AutoFuzz ensures that the maximum reward for each component is 1, adds the two components, and divides the sum by two to yield the final reward. The final reward is also constrained to the range $[0, 1]$.

$$R = \frac{\min(R_{crash}, 1) + \min(R_{cov}, 1)}{2} \tag{3}$$

## 3.5 Scheduler

The scheduler module is the core of AutoFuzz. At its essence, the scheduler partitions the entire fuzzing campaign into discrete time slots (i.e., also called rounds or cycles). Within each time slot, the scheduler employs a scheduling algorithm to select a fuzzing method for execution.

We model each fuzzing method as an arm of a non-stochastic MAB and employ the Exp3 algorithm (Auer et al., 2002) for scheduling. Compared to other algorithms like Q-learning, the stateless MAB problem is simpler and does not require state recognizing and long-term training (Sutton and Barto, 2018). Several prior studies (Yue et al., 2020, Wang et al., 2021, Wu et al., 2022, Zhang et al., 2022, Lee et al., 2023, Lin et al., 2024) have also applied MABs to fuzzing, though for different purposes, such as seed scheduling. Notably, we do not model the problem as a stochastic MAB (Wang et al., 2021, Wu et al., 2022) and use stochastic MAB algorithms like UCB1 (Auer et al., 2002), as some earlier works did (Wang et al., 2021, Wu et al., 2022). This decision stems from the interdependence among the different fuzzing methods acting as arms, because they share essential components such as the seed corpus, rendering them non-independent. For instance, as the shared seed corpus expands during the fuzzing campaign, the new coverage and crashes that all fuzzing methods (arms) are expected to find change, and the rewards that the arms receive also evolve. Additionally, their rewards may not adhere to stationary probability distributions due to the highly dynamic nature of the fuzzing process. Stochastic MAB assumes that the rewards of each arm are independently drawn from a fixed and unknown distribution (Auer et al., 2002), which does not align with our situation. In contrast, non-stochastic MAB (i.e., adversarial MAB) makes no statistical assumptions about the generation of rewards (i.e., they may be arbitrarily generated and do not follow a distribution) (Auer et al., 2002), fitting our context well. The Exp3 algorithm is a classic algorithm designed for non-stochastic MAB and has a guaranteed worst-case bound on regret (Auer et al., 2002). We also experimented and confirmed that the Exp3 algorithm performs better than UCB1 in our case.

In the Exp3 algorithm (Auer et al., 2002), a fixed parameter $\gamma \in (0, 1]$ is chosen for adjusting the weight component in probability calculation. During initialization, each arm is assigned an initial weight $w_i(1) = 1$, for $i = 1, ..., K$ ($K$ is the number of fuzzing methods here). Then, during each time slot $t$. The probabilities for each arm $i$ (i.e., fuzzing method) are calculated by

$$p_i(t) = (1-\gamma)\frac{w_i(t)}{\sum_{j=1}^{K} w_j(t)} + \frac{\gamma}{K}. \tag{4}$$

Then, the scheduler draws the arm $i_t$ according to the probabilities $p_i(t)$ (since $\sum_{i=1}^{K} p_i(t) = 1$). It selects the corresponding fuzzing method to execute, and calculates the reward $x_{i_t}(t)$. After getting the reward, the MAB needs to be updated as follows.

$$\hat{x}_j(t) = \begin{cases} x_j(t)/p_j(t) \text{ if } j = i_t, \\ \qquad\qquad 0 \text{ otherwise }. \end{cases} \tag{5}$$

$$w_j(t+1) = w_j(t)\exp(\gamma \hat{x}_j(t)/K) \tag{6}$$

We select the Exp3.S.1 algorithm from the Exp3 algorithm family because its parameters like $\gamma$ can be determined analytically, eliminating the need for parameter tuning (Auer et al., 2002, Woo et al., 2013). We empirically set the time slot length $t$ to eight minutes, trying to balance between minimizing overhead from switching arms and reducing prolonged use of suboptimal arms.

The detailed procedure of the scheduler is outlined in Algorithm 2. At first, the scheduler is initialized, which includes initializing the internal MAB and preparing the working directories. Subsequently, it initializes all fuzzing methods by sequentially launching and executing each fuzzing method for a brief period before pausing them to await scheduling. Following the initialization phase, the scheduler enters a continuous running loop. Within this loop, the scheduler first consults the MAB to choose a fuzzing method for scheduling. Subsequently, it invokes the CONTINUE method of the chosen fuzzing method to commence execution. After a predefined time interval (CycleTime), the scheduler pauses the selected fuzzing method by invoking its PAUSE method. Moreover, the scheduler triggers the reward calculation module by invoking the GETREWARD method for the recently scheduled fuzzing method and updates its reward within the MAB. This running loop persists until the user-defined SettingTime is reached.

The CONTINUE and PAUSE methods of a fuzzing method are mainly implemented by sending SIGCONT and SIGSTOP signals to the fuzzer instance, respectively. AutoFuzz does not start and stop fuzzing methods because the initialization of a fuzzer may take too much time, e.g., calibration for all seeds when starting AFL++ (Fioraldi et al., 2020). Furthermore, some fuzzers may lose their in-memory data upon restarting, as seen with MOpt's swarm data (Lyu et al., 2019). Instead, continuing and pausing fuzzing methods by sending SIGCONT and SIGSTOP signals is much faster and also simple to implement. Additionally, it's worth noting that seed synchronization among fuzzing methods occurs during the continuation of the chosen fuzzing method. Most fuzzers feature built-in synchronization methods originally intended for synchronizing multiple instances of the same fuzzer. AutoFuzz repurposes these existing methods, and sends signals to trigger instant synchronization, since existing fuzzers may only check for synchronization at fixed intervals.

An optimization in scheduling involves expediting the analysis process by concurrently conducting crash analysis and coverage analysis as soon as the selected fuzzing method commences execution. Consequently, upon the completion of the time slot, the analysis results can be promptly obtained.

**Algorithm 2** The Scheduler Procedure

```
Input: SettingTime (The time set to run by the user)
 1: InitializeScheduler()                          ▷ Initialize the scheduler (MAB)
 2: InitializeMethods()            ▷ Initialize all fuzzing methods and pause them
 3: while RunningTime < SettingTime do
 4:     ChosenMethod ← MAB.Choice()   ▷ Choose the fuzzing method with the MAB algorithm
 5:     ChosenMethod.Continue()
 6:     Sleep(CycleTime)
 7:     ChosenMethod.Pause()
 8:     MAB.UpdateReward(ChosenMethod, ChosenMethod.GetReward())
 9:     RunningTime ← RunningTime + CycleTime
10: end while
```

## 4 Evaluation

### 4.1 Implementation

We have developed a prototype of AutoFuzz in Python, consisting of approximately 1900 lines of code. As previously noted, users may prioritize different types of vulnerabilities and can assign varying weights to the reward accordingly. In our current setting, for SEGV-type vulnerabilities not recognized by sanitizers which may be hard for users to investigate the root cause further, we assign a weight of 0.7 while assigning other types of vulnerabilities a standard weight of 1. Users may customize their settings as well, e.g., giving heap vulnerabilities higher weights than stack vulnerabilities. We assign duplicated weight $W_{dup}$ in Equation 1 to be 0.1 to incentivize crash discovery. Currently, we utilize a time slot duration of eight minutes.

## 4.2 Experiment Setup

**Target programs** For our testing, we randomly select 13 programs from the UniFuzz program set (Li et al., 2021). These programs encompass real-world but not fake or front-ported vulnerabilities (Hazimeh et al., 2020, Elahi and Wang, 2024), providing a robust evaluation of the fuzzers' vulnerability detection capabilities.

**Fuzzing methods (i.e., Fuzzers and Sanitizers)** We have selected AFL++ (Fioraldi et al., 2020), libFuzzer (LLVM, 2023), and MOpt (Lyu et al., 2019) (also known as MOpt-AFL, as it is derived from AFL) as the fuzzers for our testing. AFL++ has emerged as one of the most widely utilized fuzzers in academia, particularly following the discontinuation of maintenance for AFL (Schiller et al., 2023). In contrast, libFuzzer is popular in the industry and is responsible for the discovery of many recent vulnerabilities (Google Security Team, 2018). MOpt is based on the milestone fuzzer AFL (Zalewski, 2017) and has demonstrated superior vulnerability discovery capability compared to AFL (Li et al., 2021). However, since MOpt is based on AFL and is not maintained now as well, we tested two variants, one with MOpt and one without MOpt. For example, AutoFuzz with MOpt means AutoFuzz uses all fuzzing methods including MOpt fuzzing methods during the scheduling. For each fuzzer, we experimented with three sanitizer configurations: no sanitizer, with ASan, and with UBSan.

However, due to libFuzzer's in-process fuzzing nature, distinct from traditional AFL-based fuzzers, existing programs require modifications to enable fuzzing. In the case of AFL-based fuzzers, one can directly compile the source code using AFL's wrapped compiler. Conversely, for libFuzzer, it's necessary to expose the `extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size)` function in the program to accept fuzzing input, without including a main function. To achieve this modification, we employ the following approach. Given an example program showing in Listing 2, we alter it as demonstrated in Listing 3. The test data generated by libFuzzer is then written to a temporary file, and the actual main function is invoked with the file as input. As we utilize tmpfs to host the fuzzing directory, file operations actually occur within memory, ensuring swift performance. Additionally, since the modification removes the main function, configuring programs with tools like configure will fail the validity check. Consequently, we must execute configure first, and then write a script to replace the main function.

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
    FILE *fp = fopen(argv[1], "rb");
    // ... further processing the file
}
```

Listing 2: An example program

Even using the modification method, we failed to make all the programs work to support libFuzzer. For example, if the project has multiple programs, it is hard to

```
#include <stdio.h>
extern "C"
int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
    FILE *fp = fopen("test", "wb");
    fwrite(fp, size, 1, data);
    fclose(fp);
    char *ch[] = {"./main", "test", NULL};
    main1(2, ch);
}
int main1(int argc, char *argv[]) {
    FILE *fp = fopen(argv[1], "r");
    // ... further processing the file
}
```

Listing 3: Modified program code for libFuzzer

**Table 1**: Available fuzzers and sanitizers for the programs

| | AFL++ | | | MOpt (MOpt-AFL) | | | libFuzzer | | |
|---|---|---|---|---|---|---|---|---|---|
| | Pure | ASAN | UBSAN | Pure | ASAN | UBSAN | Pure | ASAN | UBSAN |
| cflow | Yes | Yes | Yes | Yes | Yes | Yes | No | No | No |
| exiv2 | Yes | Yes | Yes | Yes | Yes | Yes | No | No | No |
| flvmeta | Yes | Yes | Yes | Yes | Yes | Yes | No | No | No |
| infotocap | Yes | Yes | Yes | Yes | Yes | Yes | No | No | No |
| jhead | Yes | Yes | Yes | Yes | Yes | Yes | No | No | No |
| jq | Yes | Yes | No | Yes | Yes | Yes | Yes | Yes | Yes |
| lame | Yes | Yes | No | Yes | Yes | Yes | No | No | No |
| mp3gain | Yes | Yes | Yes | Yes | Yes | Yes | Yes | No | Yes |
| mp42aac | Yes | Yes | Yes | Yes | Yes | Yes | No | No | No |
| mujs | Yes | Yes | Yes | Yes | Yes | Yes | No | No | No |
| pdftotext | Yes | Yes | No | Yes | Yes | Yes | No | No | No |
| tcpdump | Yes | Yes | Yes | Yes | Yes | Yes | No | No | No |
| tiffsplit | Yes | Yes | Yes | Yes | Yes | Yes | No | No | No |

ensure the whole building toolchain can still be compiled after removing the main function. Similarly, we failed to enable all the sanitizers. For example, some programs compiled with UBSan report that they have illegal assembly instructions like UD2, which is a known problem (Song et al., 2019). We list all the supported fuzzer and sanitizer combinations of the programs in Table 1.

In addition to AutoFuzz and standalone fuzzing methods, we have incorporated a straightforward Round Robin scheduling method for comparative analysis. In the Round Robin approach, each fuzzing method is scheduled consecutively for the same duration of time, and once all methods have been scheduled, the process repeats. This scheduling method resembles the approach adopted by EnFuzz (Chen et al., 2019) and Cupid (Güler et al., 2020), where each fuzzer is allocated an equal number of CPU cores.

**Experimental platform** All experiments are completed on a 64-bit machine with two 18 physical cores CPU (Intel(R) Xeon(R) Gold 6139 CPU @ 2.30GHz) and 64GB RAM. The operating system is Ubuntu 22.04.

**Evaluation metrics** In our framework, we utilize the newly acquired code coverage and the count of identified vulnerabilities in each time interval as metrics to assess the performance of each fuzzing method. Consequently, we employ these same metrics to evaluate the performance across various fuzz testing methods.

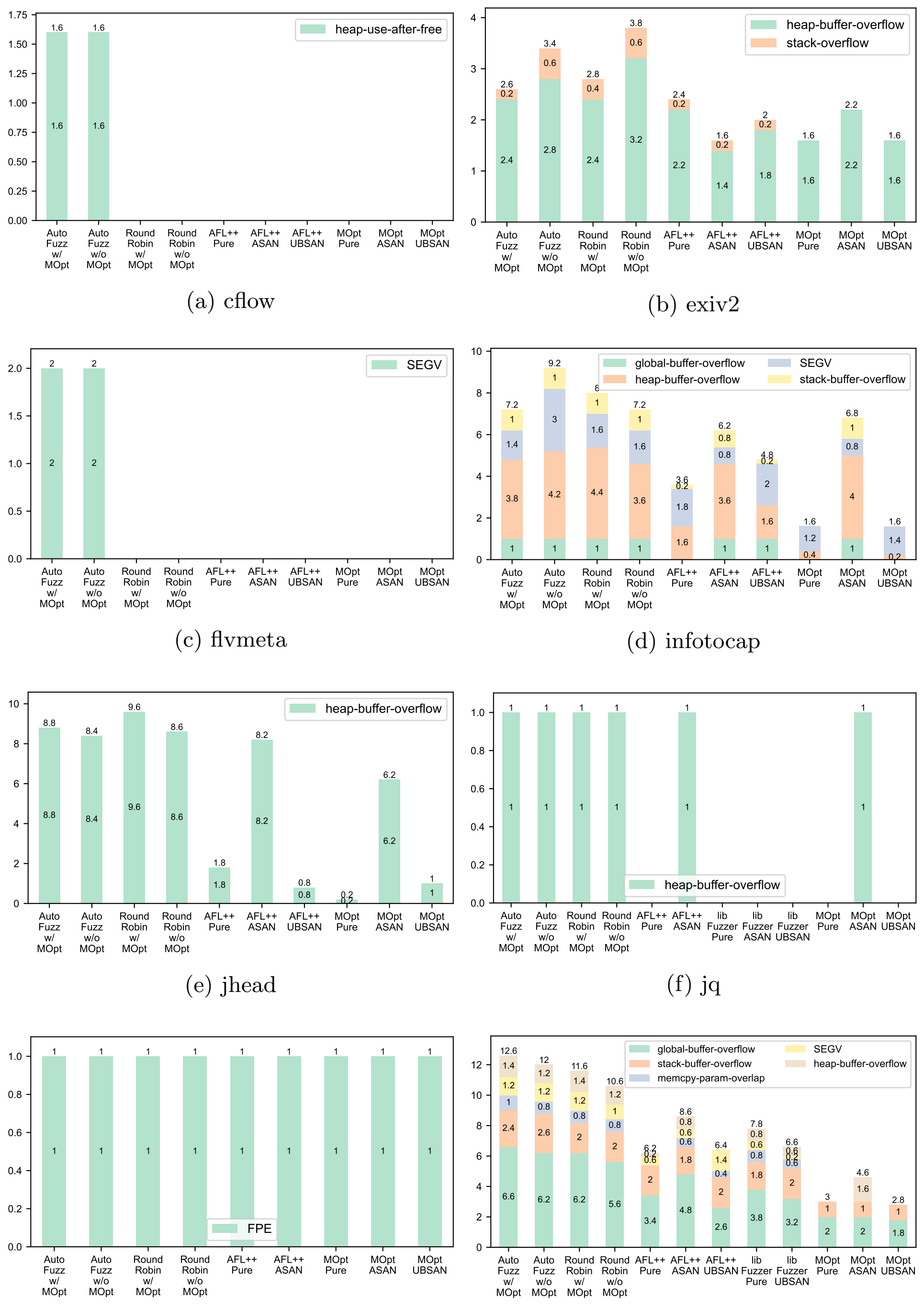
**Statistical methods** All target programs undergo 12 hours of fuzzing, and each fuzzing method is repeated five times to mitigate random effects as recommended (Klees et al., 2018).

## 4.3 Comparison of Vulnerability Found

Since the ultimate goal of fuzzing is to uncover vulnerabilities (Böhme et al., 2022). Figure 2 presents the number of vulnerabilities detected by AutoFuzz and alternative approaches.

Firstly, AutoFuzz with MOpt identifies more or the same number of vulnerabilities than any standalone fuzzing method in 12 out of 13 tested programs (except tcpdump). Similarly, AutoFuzz without MOpt performs comparably to AutoFuzz with MOpt, outperforming or matching standalone fuzzing methods in 11 out of 13 programs (except mujs and tcpdump). For instance, in cflow and flvmeta, only AutoFuzz is able to find vulnerabilities. A notable exception is tcpdump, where AutoFuzz identifies approximately 9 fewer vulnerabilities than AFL++ ASan. However, it is uncommon for a single program to contain such a large number of vulnerabilities (up to 40). Additionally, our subsequent analysis reveals that AFL++ ASan achieves significantly lower code coverage (Figure 3), suggesting that the vulnerabilities are concentrated in specific regions. Consequently, the total reward obtained by AutoFuzz in tcpdump is slightly higher on average (0.06 points) than that of AFL++ ASan (Figure 4). While standalone fuzzing methods may excel in certain programs, they may underperform in others. For instance, AFL++ ASan performs exceptionally well in mujs and tcpdump but is outperformed by AFL++ Pure in pdftotext, finding 4.4 fewer vulnerabilities. In contrast, AutoFuzz dynamically switches to more effective fuzzing methods when others do not perform well, thereby consistently achieving commendable results.

Secondly, AutoFuzz also outperforms or matches the Round Robin approach (i.e., EnFuzz&Cupid scheduling) in 10 out of 13 programs (outperforming in 6 programs and matching in 4). For example, in mp3gain, both AutoFuzz variants with and without MOpt outperform their corresponding Round Robin counterparts. Although Round Robin performs better in jhead and tcpdump, the performance gap is narrow. This is because AutoFuzz allocates more time to fuzzing methods that find more vulnerabilities, unlike the fixed-time allocation employed by Round Robin. While the Round Robin approach typically discovers more vulnerabilities than standalone fuzzing methods, such as in jq, mp3gaign, and tiffsplit, due to its ensemble style that enables fuzzing methods to complement each other, it may sometimes suffer if it includes fuzzing methods with significantly lower performance. For instance, in pdftotext, all fuzzing methods except AFL++ Pure and MOpt Pure exhibit relatively poor performance, leading to much lower results for the Round Robin approach compared to AutoFuzz



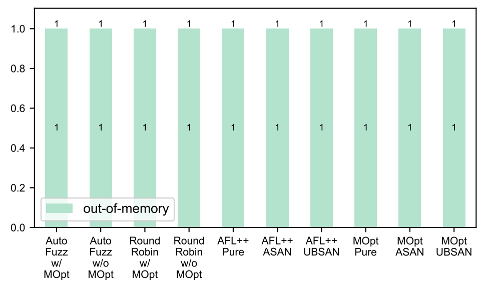
(a) cflow (b) exiv2

(c) flvmeta (d) infotocap

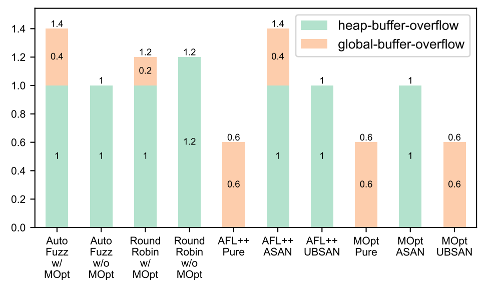
(e) jhead (f) jq

(g) lame (h) mp3gain

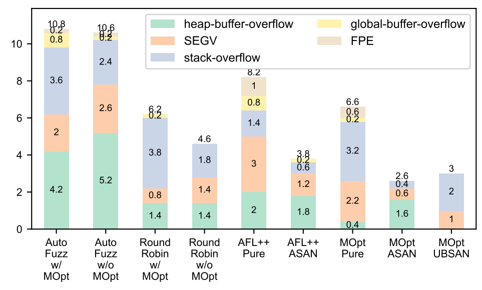
**Fig. 2**: Number of vulnerabilities found by AutoFuzz, Round Robin (i.e., EnFuzz&Cupid scheduling), and standalone fuzzing methods.



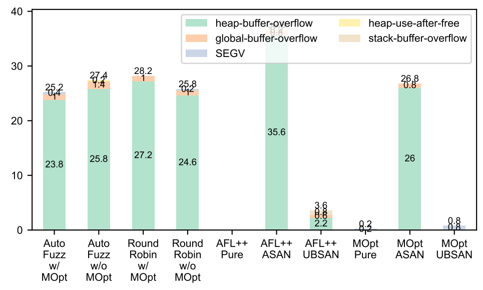
(i) mp42aac



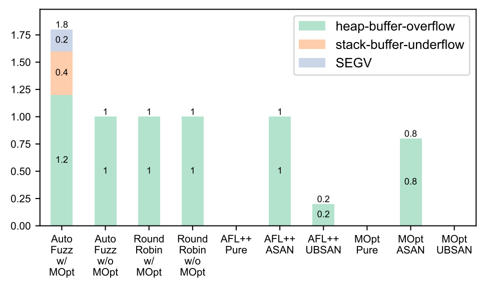
(j) mujs



(k) pdftotext



(l) tcpdump



(m) tiffsplit

**Fig. 2**: Number of vulnerabilities found by AutoFuzz, Round Robin (i.e., EnFuzz&Cupid scheduling), and standalone fuzzing methods. (Cont.)

(finding up to 6 fewer vulnerabilities). AutoFuzz does not suffer from such drawbacks, as its scheduling is dynamically adjusted according to reward feedback.

Additionally, we observe that sanitizers play a crucial role in detecting certain vulnerabilities. For example, in both jq and tiffsplit, a heap-buffer-overflow vulnerability could only be detected when ASan is enabled (e.g., AFL++ ASan and MOpt ASan). If users arbitrarily decide not to use sanitizers during fuzzing, they may miss detecting such vulnerabilities.

## 4.4 Comparison of Code Coverage

Code coverage is also a common metric for fuzzer comparison, and we have illustrated the growth of code coverage over time for all test programs in Figure 3. It is evident that AutoFuzz consistently achieves commendable code coverage across all programs, ranking as the best fuzzer in 8 out of 13 programs and within the top three in the remaining 5 programs. Notably, in mp3gain, tcpdump, and tiffsplit, AutoFuzz's code coverage significantly surpasses that of standalone fuzzing methods and the Round Robin method. Although AutoFuzz may not outperform the top fuzzing method in lame, pdftotext, jq, and mp42aac, the disparity is marginal (e.g., 99.93% for lame, 99.38% for pdftotext, 99.07% for jq, and 97.45% for mp42aac in the version without MOpt). Overall, AutoFuzz demonstrates a significant deviation from the performance of the poorest standalone fuzzing method (e.g., AFL++ ASan in pdftotext). These findings suggest that although code coverage may not be AutoFuzz's primary objective, it consistently achieves satisfactory code coverage even when the effectiveness of standalone fuzzing methods is uncertain before testing. Moreover, AutoFuzz's code coverage is comparable to or exceeds that of Round Robin in all the programs, indicating that employing MAB to optimize fuzzing method scheduling outperforms the blind round-robin scheduling approach.

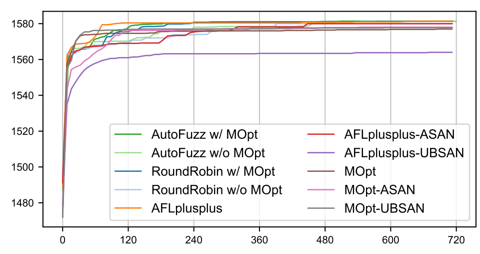
## 4.5 Comparison of Final Reward

**Table 2**: The comparison of total rewards of different approaches.

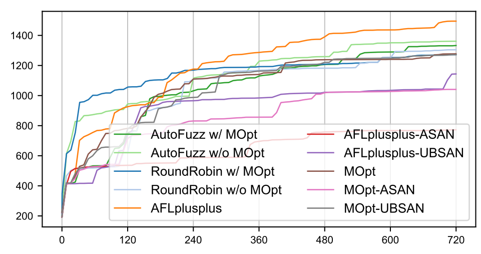
| | AutoFuzz | | Round Robin | | AFL++ | | | MOpt | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | w/[a] | w/o[b] | w/[a] | w/o[b] | Pure | ASAN | UBSAN | Pure | ASAN | UBSAN |
| cflow | 29.853 | 29.853 | 29.053 | 29.027 | 29.053 | 29.027 | 28.733 | 28.972 | 28.990 | 28.987 |
| exiv2 | 7.719 | 8.256 | 7.528 | 8.182 | 8.399 | 4.520 | 6.508 | 6.953 | 6.113 | 6.925 |
| flvmeta | 7.805 | 7.805 | 7.105 | 7.101 | 7.101 | 7.105 | 7.101 | 7.088 | 7.078 | 7.091 |
| infotocap | 27.385 | 30.339 | 27.768 | 26.515 | 26.894 | 26.572 | 28.092 | 23.119 | 24.194 | 24.564 |
| jhead | 17.705 | 17.505 | 18.105 | 17.605 | 14.185 | 17.405 | 13.705 | 13.115 | 16.175 | 13.545 |
| jq | 35.938 | 36.092 | 35.721 | 35.841 | 35.917 | 36.163 | | 35.515 | 35.580 | 35.522 |
| lame | 3.494 | 3.494 | 3.496 | 3.485 | 3.494 | 3.496 | | 3.494 | 3.494 | 3.494 |
| mp3gain | 45.827 | 45.492 | 44.902 | 44.432 | 41.783 | 42.522 | 39.699 | 38.243 | 37.481 | 38.429 |
| mp42aac | 9.160 | 9.463 | 9.168 | 9.189 | 9.539 | 9.039 | 9.632 | 9.121 | 8.513 | 9.055 |
| mujs | 44.751 | 44.905 | 44.675 | 44.770 | 44.643 | 43.634 | 44.821 | 43.432 | 42.302 | 43.597 |
| pdftotext | 26.546 | 26.248 | 24.108 | 23.027 | 24.886 | 22.032 | | 24.442 | 21.430 | 22.162 |
| tcpdump | 51.306 | 52.915 | 52.686 | 51.246 | 38.616 | 51.554 | 38.580 | 37.470 | 43.425 | 37.120 |
| tiffsplit | 20.084 | 19.520 | 18.098 | 18.356 | 18.067 | 17.321 | 17.909 | 18.269 | 14.061 | 17.111 |
| Sum | 327.571 | 331.888 | 322.412 | 318.776 | 302.575 | 310.390 | | 289.233 | 288.836 | 287.602 |

[a] With MOpt [b] Without MOpt

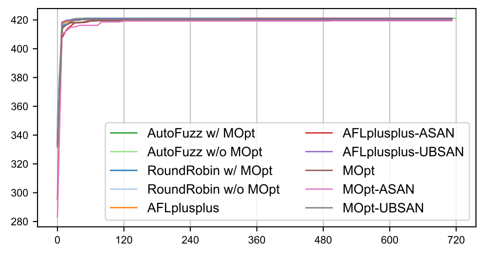
For considering both the discovered vulnerability and code coverage at the same time, we have analyzed the rewards obtained by various approaches. We compute the final reward for each approach using the equations outlined in Section 3.4, with $W_{dup}$ set to 0 to indicate that duplicated crashes yield no reward. The value of $W_{dup}$ here is different from its value during fuzzing (i.e., a non-zero value to incentivize the fuzzing methods of AutoFuzz to discover crashes), but this adjustment reflects the reality that only unique vulnerabilities ultimately count (Klees et al., 2018). Figure 4 presents



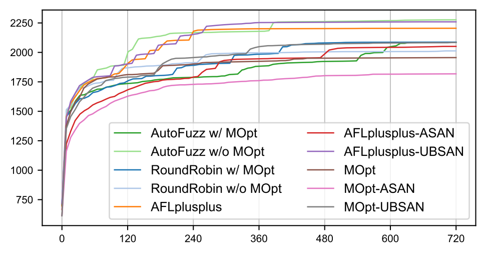
(a) cflow



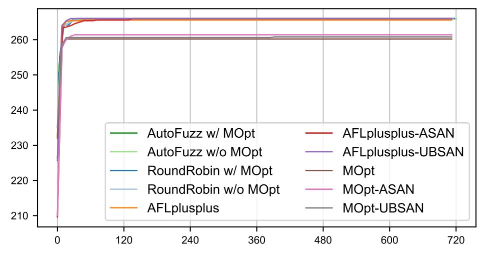
(b) exiv2



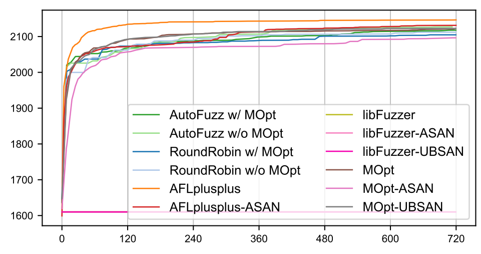
(c) flvmeta



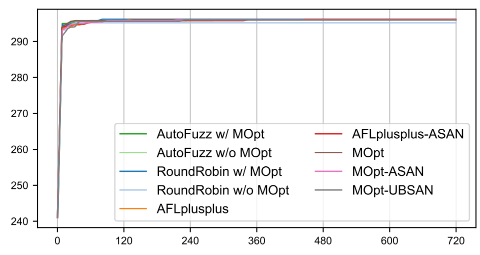
(d) infotocap



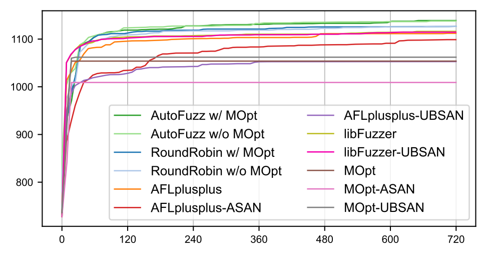
(e) jhead



(f) jq

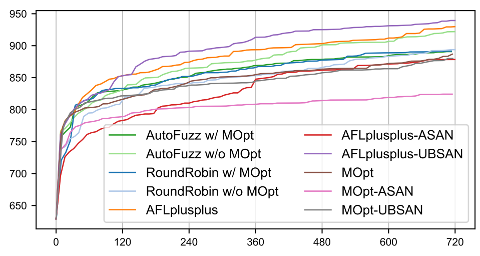


(g) lame

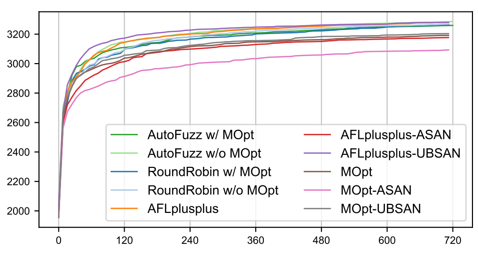


(h) mp3gain

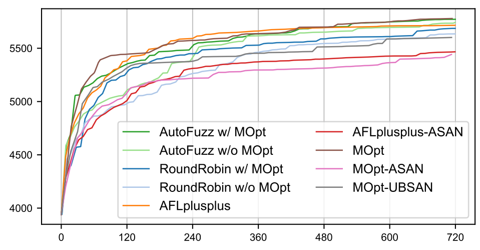
**Fig. 3**: Branch coverage growth of different approaches.



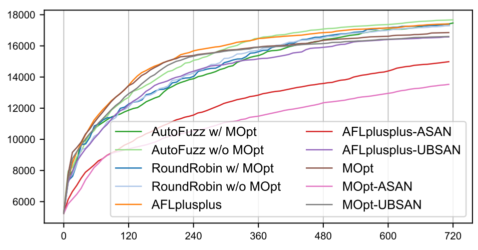
(i) mp42aac



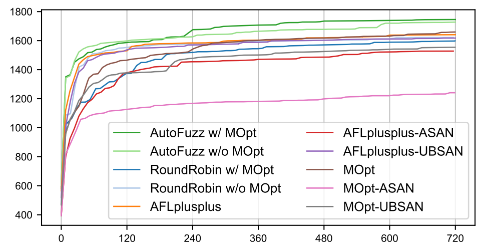
(j) mujs



(k) pdftotext



(l) tcpdump

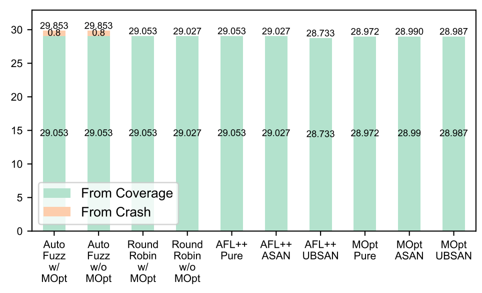


(m) tiffsplit

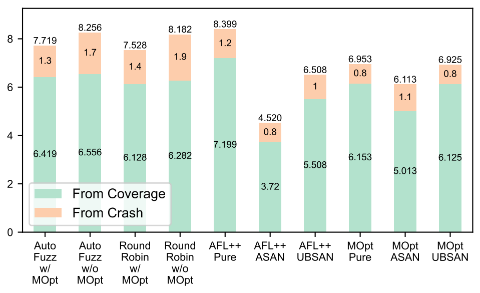
**Fig. 3**: Branch coverage growth of different approaches. (Cont.)

the breakdown of the final rewards for different approaches. Notably, code coverage is a dominant factor in the reward computation, primarily due to the infrequent occurrence of vulnerability discoveries. This observation aligns with real-world fuzzing campaigns, where vulnerabilities are often scarce, particularly in actively maintained PUTs. Higher code coverage also increases the likelihood of uncovering vulnerabilities (Klees et al., 2018, Böhme et al., 2022).

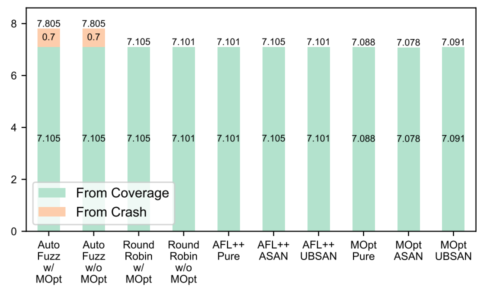
The total rewards accumulated from vulnerability discoveries are also summarized in Table 2 ($W_{dup}$ is set to 0 as well), and the p-values when comparing AutoFuzz with other approaches using the Mann Whitney U-test (Klees et al., 2018) are shown in Table 3. Notably, certain standalone fuzzing methods are not applicable to all programs and are therefore excluded from the tally. It is evident that AutoFuzz achieves the highest total reward compared to Round Robin and standalone fuzzing methods, underscoring the effectiveness of the Exp3 scheduling algorithm in autonomously



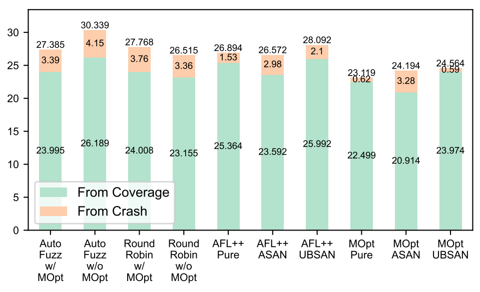
(a) cflow



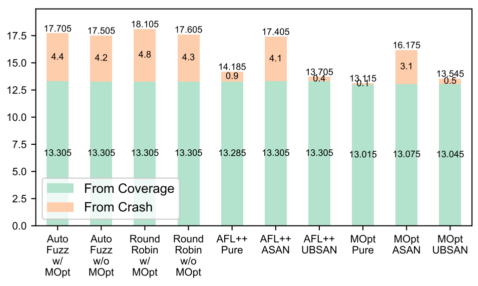
(b) exiv2



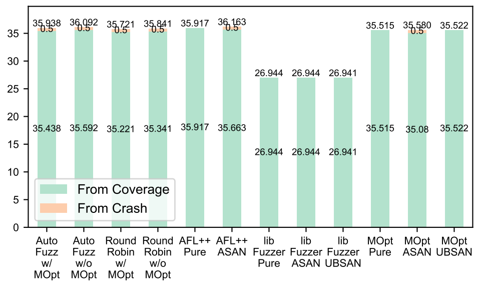
(c) flvmeta



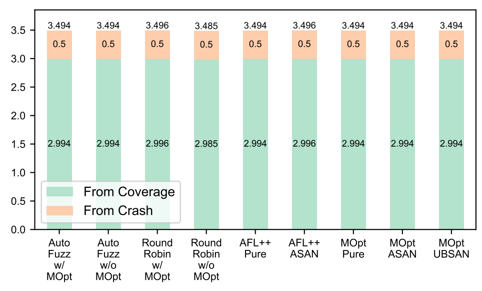
(d) infotocap



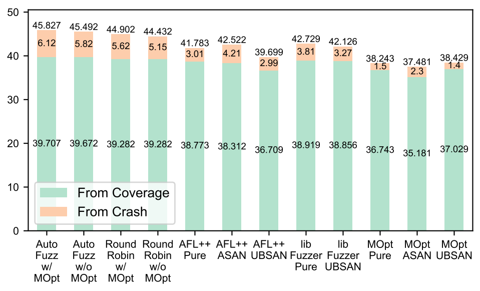
(e) jhead



(f) jq

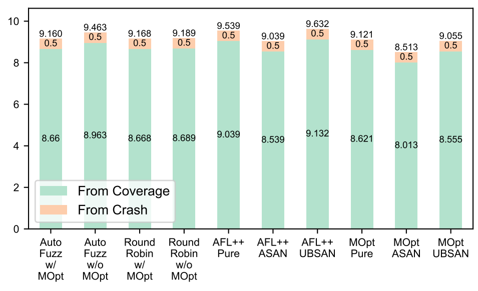


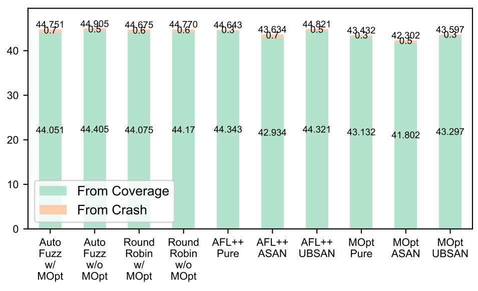
(g) lame

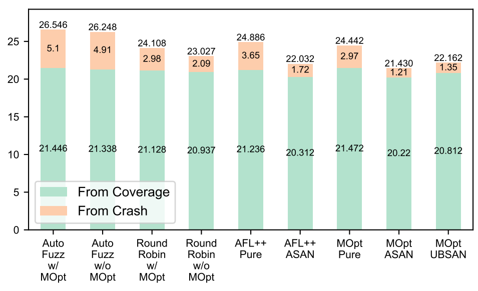


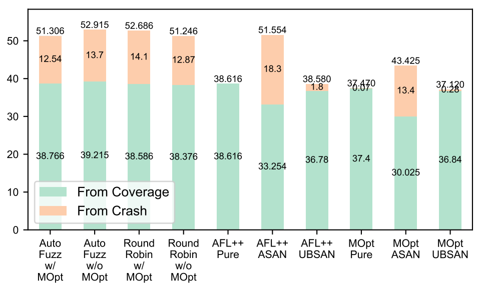
(h) mp3gain

**Fig. 4**: The breakdown of the final rewards of different approaches.


(i) mp42aac


(j) mujs


(k) pdftotext


(l) tcpdump

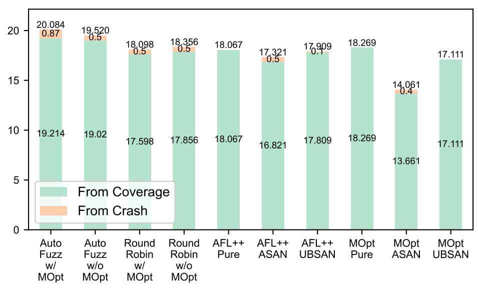

(m) tiffsplit

**Fig. 4**: The breakdown of the final rewards of different approaches. (Cont.)

selecting suitable fuzzing methods to maximize overall reward. Most of the p-values presented in Table 3 are below 0.05, indicating that the differences between AutoFuzz and other approaches are statistically significant. The only exception is the comparison involving AutoFuzz with MOpt and Round Robin with MOpt, where the p-value exceeds 0.05. This is because AutoFuzz with MOpt achieves a lower total reward compared to AutoFuzz without MOpt, whereas Round Robin with MOpt yields a higher total reward than Round Robin without MOpt, narrowing the gap between them. The higher total reward for Round Robin with MOpt can be attributed to its discovery of more crashes, unlike AutoFuzz with MOpt, which does not find more crashes

**Table 3**: The p-values of total rewards when comparing AutoFuzz with other approaches.

| | AutoFuzz with MOpt | AutoFuzz without MOpt |
|---|---|---|
| Round Robin with MOpt | 0.151 | 0.016 |
| Round Robin without MOpt | 0.016 | 0.008 |
| AFL++ Pure | 0.008 | 0.008 |
| AFL++ ASAN | 0.008 | 0.008 |
| MOpt Pure | 0.008 | 0.008 |
| MOpt ASAN | 0.008 | 0.008 |
| MOpt UBSAN | 0.008 | 0.008 |

than AutoFuzz without MOpt and we guess it is because AutoFuzz already has good enough ensemble capability when scheduling other high-performing fuzzers.

### 4.6 Scheduled Time in AutoFuzz

To explore how AutoFuzz schedules different fuzzing methods, we present the total time allocated to each fuzzing method in Figure 5. It shows that arms with higher rewards usually receive more scheduled time. For instance, in jq, AutoFuzz without MOpt, allocates AFL++ Pure and AFL++ ASan more time compared to libFuzzer-based fuzzing methods, because libFuzzer-based fuzzing methods do get lower rewards as shown in Figure 4. Similarly, in cflow, AutoFuzz without MOpt allocates AFL++ Pure more time than AFL++ UBSan since the former method gets more rewards in Figure 4. These findings indicate that our MAB scheduling algorithm is effectively functioning and meeting our expectations.

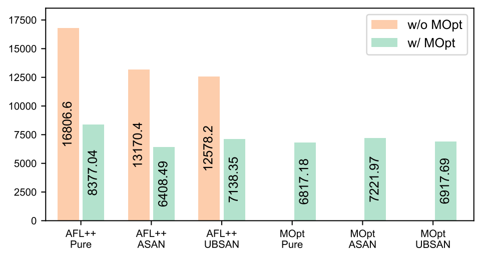
## 5 Discussion

### 5.1 Threats to Validity

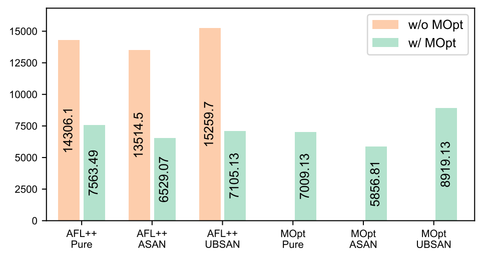
Now we use 13 programs together with their initial seeds from the UniFuzz benchmark for the evaluation. There are other benchmarks like Magma (Hazimeh et al., 2020) and FuzzBench (Metzman et al., 2021) we have not tested on. Magma (Hazimeh et al., 2020) uses a front-ported technique to increase the number of vulnerabilities in programs, and FuzzBench (Metzman et al., 2021) more focuses on code coverage. Although we believe the design of AutoFuzz is general enough, using diverse benchmarks will make the evaluation results more convincing. However, it is not affordable for us now since we need to test about 10 fuzzing methods for each PUT, and we leave that as future work. Similarly, our implementation of AutoFuzz already can scale to multiple CPU cores, but we limit it to a single CPU core for evaluating existing approaches with affordable resources.

### 5.2 Limitations

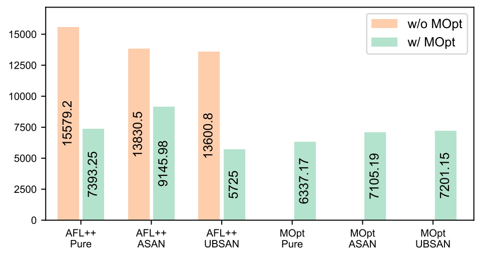
We should not add too many fuzzing methods in AutoFuzz. This is because an excessive number of methods can prolong the learning process for the Exp3 algorithm to identify arms with higher rewards. To address this challenge, we propose two strategies. Firstly, we advocate for a selective approach in choosing fuzzers, prioritizing



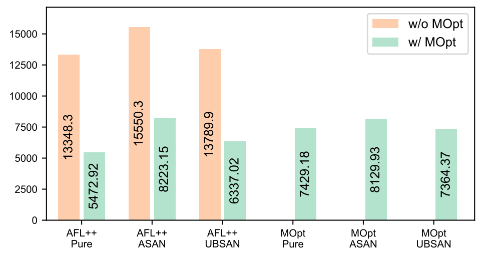
(a) cflow



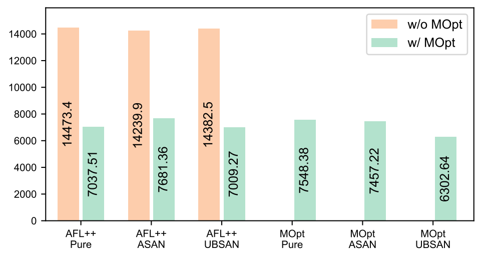
(b) exiv2



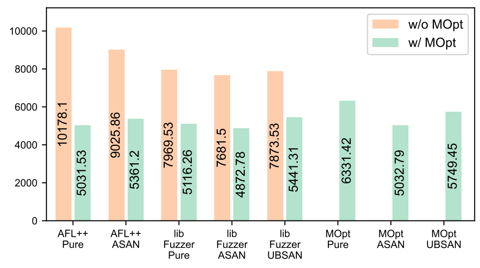
(c) flvmeta



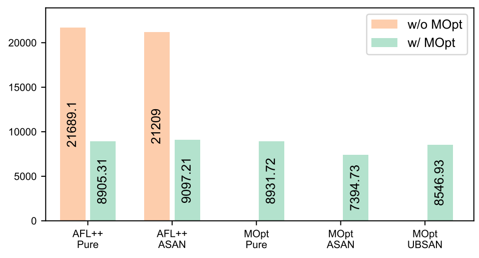
(d) infotocap



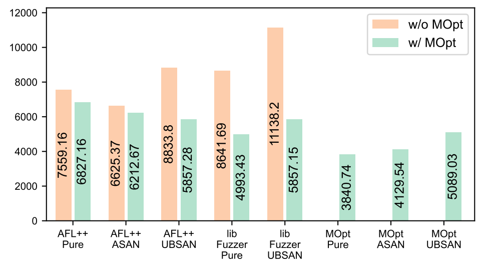
(e) jhead



(f) jq

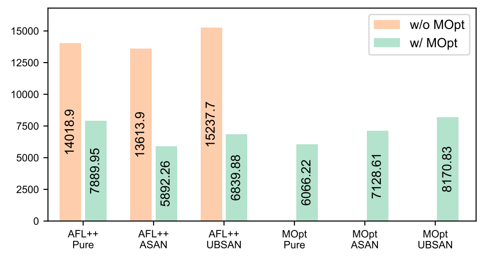


(g) lame

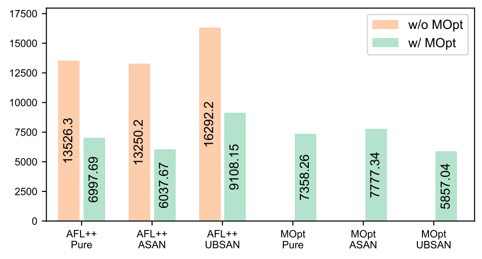


(h) mp3gain

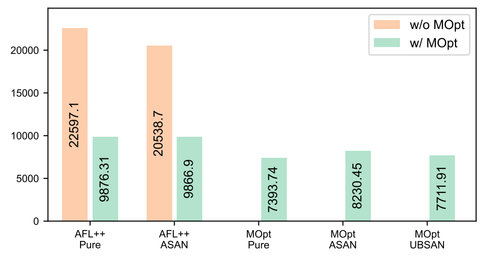
**Fig. 5**: The scheduled time of different fuzzing methods in AutoFuzz (in seconds).



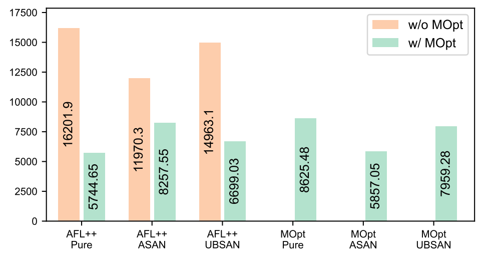
(i) mp42aac



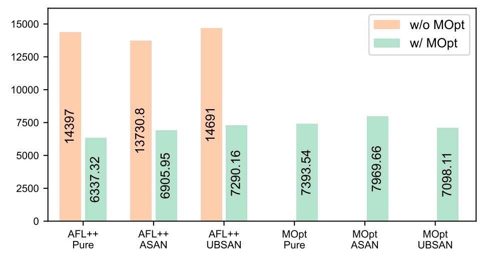
(j) mujs



(k) pdftotext



(l) tcpdump



(m) tiffsplit

**Fig. 5**: The scheduled time of different fuzzing methods in AutoFuzz (in seconds). (Cont.)

those with robust performance and aiming for diversity among them (Chen et al., 2019). Secondly, not all sanitizers need to be paired with every fuzzer to create fuzzing methods; rather, they can only be paired with fuzzers demonstrating relatively stable performance. This strategy assumes that the fuzzers could mutate the shared seeds among fuzzing methods to uncover vulnerabilities. One exception is for fuzzers tightly integrated with sanitizers, such as ParmeSan (Österlund et al., 2020), which depend on the use of sanitized PUTs.

## 6 Related Work

**Coverage-guided fuzzing** Coverage-guided fuzzing (CGF) (Zalewski, 2017, LLVM, 2023, Lyu et al., 2019, Lemieux and Sen, 2018, Yue et al., 2020) is a widely employed technique for uncovering vulnerabilities in software (Manes et al., 2019, Zhu et al., 2022, Mallissery and Wu, 2023, Google Security Team, 2018, Li et al., 2021). This approach systematically explores the state space of the PUT by tracking code coverage and incorporating new inputs that yield previously unexplored coverage into the seed pool for subsequent fuzzing iterations. Additionally, sanitizers like ASan (Serebryany et al., 2012) can be leveraged to instrument the PUT, facilitating the immediate detection of various vulnerabilities by triggering PUT crashes when vulnerabilities occur (e.g., even a single-byte stack buffer overflow can be detected) (Serebryany et al., 2012, Song et al., 2019), albeit with some program execution slowdown (e.g., 2x for ASan (Serebryany et al., 2012)). Numerous CGF fuzzers have been proposed; however, research indicates that no single CGF fuzzer performs optimally across all programs (Li et al., 2021, Hazimeh et al., 2020, Metzman et al., 2021, Liu et al., 2023). Moreover, studies have shown that employing multiple types of fuzzers in tandem, known as ensemble-style fuzzing, can outperform single-fuzzer approaches (Chen et al., 2019).

**Multiple fuzzer scheduling** Several methods have been proposed for scheduling multiple fuzzers. Hybrid fuzzers like Driller (Stephens et al., 2016, Yun et al., 2018) usually combine the symbolic execution engine with a CGF fuzzer to better explore the program. The scheduling strategy often involves running the CGF fuzzer continuously and activating the symbolic execution engine when the CGF fuzzer encounters a roadblock, or simultaneously initiating both of them. EnFuzz (Chen et al., 2019) is the first method to intentionally use several different kinds of fuzzers to fuzz a PUT at the same time (called ensemble fuzzing). However, the fuzzer combinations in both hybrid fuzzers and EnFuzz are manually selected and fixed (e.g., two fuzzers in Driller, and four fuzzers in EnFuzz). In Cupid (Güler et al., 2020), the authors propose to automatically select fuzzer combinations for programs by estimating the coverage probabilities of different combinations of fuzzers. Nevertheless, Cupid does not consider sanitizer usage and cannot allocate different timeshares (or CPU cores) to selected fuzzers. CollabFuzz (Österlund et al., 2021) focuses on optimizing test case scheduling among multiple fuzzers, disregarding fuzzer selection. $\mu$Fuzz (Chen et al., 2023) utilizes a microservice-based architecture to redesign fuzzers, enabling efficient parallel fuzzing. The enhancements introduced by CollabFuzz and $\mu$Fuzz are orthogonal to our approach.

**Sanitizer usage in fuzzing** Sanitizer usage in fuzzing can be categorized into three types. The first type is only using sanitizers in post-processing. Since sanitizers bring in extra overhead, users may not use sanitizers during fuzzing (Manes et al., 2019, Lyu et al., 2019, Jauernig et al., 2023) for a higher fuzzing speed and fully rely on the operating system and compiler built-in mechanisms (like stack canary (Cowan et al., 1998)) to detect vulnerabilities. Sanitizers like ASan are only employed post-fuzzing for tasks such as vulnerability deduplication (Li et al., 2021). The second type entails using sanitizers to instrument the PUT during fuzzing. For example, in OSS-Fuzz (Google Security Team, 2018), each kind of sanitizer has a corresponding fuzzer instance, which certainly requires significant resources. The third type indirectly leverages sanitizer information for fuzzing (Österlund et al., 2020, Zheng et al., 2023). For example, ParmeSan (Österlund et al., 2020) utilizes sanitizers to identify interesting targets and guide the fuzzer toward them. AutoFuzz falls under the second type; however, it dynamically switches between sanitized and unsanitized PUTs for optimal overall performance.

**MAB usage in fuzzing** Several studies also employ multi-armed bandit (MAB) models to enhance fuzzing techniques. In EcoFuzz (Yue et al., 2020), researchers model the process of seed searching and energy allocation as a variant of the Adversarial Multi-Armed Bandit (AMAB). They treat seeds as arms and also make no statistical assumptions about rewards. They design custom algorithms for seed selection and energy assignment. AFL-HIER (Wang et al., 2021) organizes seeds in a tree structure, where each node's subtrees are modeled as the arms of an MAB, and UCB1 is used to select which subtree to fuzz. In BanditFuzz (Scott et al., 2021), Thompson sampling serves as the MAB algorithm. Two MABs operate at different levels: one selects between the mutator and the generator, while the other chooses between inserting and replacing grammatical constructs of the SMT-LIB language. Havoc$_{MAB}$ (Wu et al., 2022) also employs MABs at two levels: a stacking size-level bandit to select the stacking size to apply, and a mutator-level bandit to choose between chunk mutators and unit mutators, with UCB1-Tuned applied at both levels. MobFuzz (Zhang et al., 2022) utilizes a multi-player multi-armed bandit model to handle the selection of multiple objective combinations, such as memory consumption and the number of satisfied comparison bytes, employing UCB1. SeamFuzz (Lee et al., 2023) uses MAB to select mutation methods, specifying where and how to mutate, with Thompson sampling as the selection algorithm. Marco (Hu et al., 2024) uses Thompson sampling to estimate the transition probability of a code branch based on the solving results of corresponding path constraints. HyperGo (Lin et al., 2024) models seeds as the arms of an MAB, and integrates a custom reward to the power schedule of directed fuzzing. AutoFuzz is different from these approaches since it uses the MAB for a different purpose and models fuzzer instances but not lower-level things like seeds as the arms of the MAB.

## 7 Conclusion

Despite the effectiveness of CGF fuzzers, selecting the optimal fuzzer or combination of fuzzers for a given particular PUT remains challenging, given the absence of a one-size-fits-all solution. Additionally, determining whether to enable sanitizers and which sanitizers to enable during fuzzing raises questions. To address these challenges, we introduce AutoFuzz, a novel approach employing a multi-armed bandit scheme to automatically schedule different fuzzing methods (a combination of fuzzer and sanitizer) at runtime. Specifically, AutoFuzz models different fuzzing methods as arms of a non-stochastic MAB and uses Exp3 as the scheduling algorithm. Our prototype demonstrates it could find more vulnerabilities, and also receive higher rewards than any standalone fuzzing method and or a Round Robin (EnFuzz&Cupid scheduling) scheduling solution. We believe that AutoFuzz does not require extensive experiments and could serve as a practical solution for selecting fuzzers and sanitizers in reality.

**Author contributions** Yijia Gao: writing original draft, methodology, software, visualization, and experiment analysis. Wenrui Zeng: software, formal analysis. Siyuan Liu: software. Yingpei Zeng: review & editing, supervision, project administration, funding acquisition. All authors reviewed the manuscript.

**Funding** This research was partially funded by the Zhejiang Provincial Natural Science Foundation of China under Grant No. LY22F020022, the National Natural Science Foundation of China under Grant No. 61902098.

**Data availability** No datasets were generated or analyzed during the current study. The code is going to open source at https://github.com/AutoFuzz/AutoFuzz.

## Declarations

**Competing interests** The authors declare no competing interests.

## References

Zalewski, M.: AFL - American Fuzzy Lop (2017). http://lcamtuf.coredump.cx/afl/ Accessed 2024-05-06

Böhme, M., Pham, V.-T., Roychoudhury, A.: Coverage-based Greybox Fuzzing as Markov Chain. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 1032–1043. ACM, Vienna (2016). https://doi.org/10.1145/2976749.2978428

Miller, B.P., Fredriksen, L., So, B.: An Empirical Study of the Reliability of UNIX Utilities. Communications of the ACM **33**(12), 32–44 (1990) https://doi.org/10.1145/96267.96279

Manes, V.J.M., Han, H.S., Han, C., Cha, Egele, M., Schwartz, E.J., Woo, M.: The Art, Science, and Engineering of Fuzzing: A Survey. IEEE Transactions on Software Engineering, 1–21 (2019) https://doi.org/10.1109/TSE.2019.2946563 arXiv:1812.00140

Zhu, X., Wen, S., Camtepe, S., Xiang, Y.: Fuzzing: A Survey for Roadmap. ACM Computing Surveys, 1–34 (2022) https://doi.org/10.1145/3512345

Mallissery, S., Wu, Y.-S.: Demystify the fuzzing methods: A comprehensive survey. ACM Computing Surveys **56**(3), 1–38 (2023)

Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: AddressSanitizer: A fast address sanity checker. In: USENIX Annual Technical Conference (ATC), pp. 309–318 (2012)

Google Security Team: A New Chapter for OSS-Fuzz (2018). https://security.googleblog.com/2018/11/a-new-chapter-for-oss-fuzz.html Accessed 2024-05-06

LLVM: libFuzzer - a library for coverage-guided fuzz testing (2023). http://llvm.org/docs/LibFuzzer.html Accessed 2024-05-06

Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: AFL++: Combining incremental steps of fuzzing research. WOOT 2020 - 14th USENIX Workshop on Offensive Technologies, co-located with USENIX Security 2020 (2020)

Google: Honggfuzz (2023). https://github.com/google/honggfuzz Accessed 2024-05-06

Google: syzkaller - kernel fuzzer (2015). https://github.com/google/syzkaller Accessed 2024-05-06

Pan, G., Lin, X., Zhang, X., Jia, Y., Ji, S., Wu, C., Ying, X., Wang, J., Wu, Y.: V-Shuttle: Scalable and Semantics-Aware Hypervisor Virtual Device Fuzzing. Proceedings of the ACM Conference on Computer and Communications Security (CCS), 2197–2213 (2021) https://doi.org/10.1145/3460120.3484811

Liu, Q., Toffalini, F., Zhou, Y., Payer, M.: Videzzo: Dependency-aware virtual device fuzzing. In: 2023 IEEE Symposium on Security and Privacy (SP), pp. 3228–3245 (2023). IEEE Computer Society

Pham, V.-t., Böhme, M., Roychoudhury, A.: AFLNet: A Greybox Fuzzer for Network Protocols. In: Proceedings of the 13rd IEEE International Conference on Software Testing, Verification and Validation : Testing Tools Track, pp. 460–465 (2020). https://doi.org/10.1109/ICST46399.2020.00062

Andronidis, A., Cadar, C.: Snapfuzz: high-throughput fuzzing of network applications. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 340–351 (2022)

Ba, J., Böhme, M., Mirzamomen, Z., Roychoudhury, A.: Stateful Greybox Fuzzing. In: Proceedings of the 31st USENIX Security Symposium (2022). http://arxiv.org/abs/2204.02545

Liang, Y., Liu, S., Hu, H.: Detecting logical bugs of {DBMS} with coverage-based guidance. In: Proceedings of the 31st USENIX Security Symposium, pp. 4309–4326 (2022)

Jiang, Z.-M., Bai, J.-J., Su, Z.: Dynsql: Stateful fuzzing for database management systems with complex and valid sql query generation. In: Proceedings of 32nd USENIX Security Symposium (2023)

Zeng, Y., Zhu, F., Zhang, S., Yang, Y., Yi, S., Pan, Y., Xie, G., Wu, T.: Dafuzz: data-aware fuzzing of in-memory data stores. PeerJ Computer Science **9**, 1592 (2023)

Meng, R., Pîrlea, G., Roychoudhury, A., Sergey, I.: Greybox fuzzing of distributed systems. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 1615–1629 (2023)

Chen, Y.: Chronos: Finding timeout bugs in practical distributed systems by deep-priority fuzzing with transient delay. In: 2024 IEEE Symposium on Security and Privacy (SP), pp. 109–109 (2024). IEEE Computer Society

Zheng, Y., Davanian, A., Yin, H., Song, C., Zhu, H., Sun, L.: FIRM-AFL: High-throughput greybox fuzzing of IoT firmware via augmented process emulation. In: Proceedings of the 28th USENIX Security Symposium, pp. 1099–1114 (2019)

Zeng, Y., Lin, M., Guo, S., Shen, Y., Cui, T., Wu, T., Zheng, Q., Wang, Q.: Multifuzz: A coverage-based multiparty-protocol fuzzer for IoT publish/subscribe protocols. Sensors **20**(18), 1–19 (2020) https://doi.org/10.3390/s20185194

Zhu, X., Liu, S., Jolfaei, A.: A fuzzing method for security testing of sensors. IEEE Sensors Journal (2023)

Li, Y., Ji, S., Chen, Y., Liang, S., Lee, W.H., Chen, Y., Lyu, C., Wu, C., Beyah, R., Cheng, P., Lu, K., Wang, T.: UNIFUZZ: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In: Proceedings of the 30th USENIX Security Symposium (2021)

Hazimeh, A., Herrera, A., Payer, M.: Magma: A ground-truth fuzzing benchmark. Proc. ACM Meas. Anal. Comput. Syst. **4**(3) (2020) https://doi.org/10.1145/3428334

Metzman, J., Szekeres, L., Simon, L., Sprabery, R., Arya, A.: 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, pp. 1393–1403 (2021). https://doi.org/10.1145/3468264.3473932

Liu, D., Metzman, J., Bohme, M., Chang, O., Arya, A.: Sbft tool competition 2023 - fuzzing track. In: 2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT), pp. 51–54. IEEE Computer Society, Los Alamitos, CA, USA (2023). https://doi.org/10.1109/SBFT59156.2023.00016

Chen, Y., Jiang, Y., Ma, F., Liang, J., Wang, M., Zhou, C., Jiao, X., Su, Z.: Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In: Proceedings of the 28th USENIX Security Symposium, pp. 1967–1983 (2019)

Güler, E., Görz, P., Geretto, E., Jemmett, A., Österlund, S., Bos, H., Giuffrida, C., Holz, T.: Cupid : Automatic Fuzzer Selection for Collaborative Fuzzing. In: Annual Computer Security Applications Conference (ACSAC), pp. 360–372 (2020). https://doi.org/10.1145/3427228.3427266

Song, D., Lettner, J., Rajasekaran, P., Na, Y., Volckaert, S., Larsen, P., Franz, M.: Sok: Sanitizing for security. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 1275–1295 (2019). IEEE

Lyu, C., Ji, S., Zhang, C., Li, Y., Lee, W.H., Song, Y., Beyah, R.: MOPT: Optimized mutation scheduling for fuzzers. In: Proceedings of the 28th USENIX Security Symposium, pp. 1949–1966 (2019)

Jauernig, P., Jakobovic, D., Picek, S., Stapf, E., Sadeghi, A.-R.: DARWIN: Survival of the Fittest Fuzzing Mutators. In: NDSS (2023). https://doi.org/10.14722/ndss.2023.23159

Auer, P., Cesa-Bianchi, N., Freund, Y., Schapire, R.E.: The nonstochastic multiarmed bandit problem. SIAM journal on computing **32**(1), 48–77 (2002)

LLVM: UndefinedBehaviorSanitizer (2023). https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html Accessed 2024-05-06

Lemieux, C., Sen, K.: Fairfuzz: A targeted mutation strategy for increasing Greybox fuzz testing coverage. ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 475–485 (2018) https://doi.org/10.1145/3238147.3238176

Yue, T., Wang, P., Tang, Y., Wang, E., Yu, B., Lu, K., Zhou, X.: EcoFuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. Proceedings of the 29th USENIX Security Symposium, 2307–2324 (2020)

Klees, G., Ruef, A., Cooper, B., Wei, S., Hicks, M.: Evaluating fuzz testing. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS), pp. 2123–2138. ACM Press, Toronto (2018). https://doi.org/10.1145/3243734.3243804

Böhme, M., Szekeres, L., Metzman, J.: On the reliability of coverage-based fuzzer benchmarking. In: Proceedings of the 44th International Conference on Software Engineering (ICSE), pp. 1621–1633 (2022)

Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. Machine learning **47**(2), 235–256 (2002)

Wang, J., Song, C., Yin, H.: Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. In: NDSS (2021)

Wu, M., Jiang, L., Xiang, J., Huang, Y., Cui, H., Zhang, L., Zhang, Y.: One fuzzing strategy to rule them all. In: Proceedings of the 44th International Conference on Software Engineering (ICSE), pp. 1634–1645 (2022)

Zhang, G., Wang, P., Yue, T., Kong, X., Huang, S., Zhou, X., Lu, K.: Mobfuzz:

Adaptive multi-objective optimization in gray-box fuzzing. In: NDSS (2022)

Lee, M., Cha, S., Oh, H.: Learning seed-adaptive mutation strategies for greybox fuzzing. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 384–396 (2023). IEEE

Lin, P., Wang, P., Zhou, X., Xie, W., Lu, K., Zhang, G.: Hypergo: Probability-based directed hybrid fuzzing. Computers & Security **142**, 103851 (2024)

Lin, M., Zeng, Y., Wu, T., Wang, Q., Fang, L., Guo, S.: GSA-Fuzz: Optimize Seed Mutation with Gravitational Search Algorithm. Security and Communication Networks **2022** (2022) https://doi.org/10.1155/2022/1505842

Chen, Y., Zhong, R., Yang, Y., Hu, H., Wu, D., Lee, W.: $\mu$fuzz: Redesign of parallel fuzzing using microservice architecture. In: Proceedings of the 32nd USENIX Security Symposium (2023)

Ng, A.Y., Harada, D., Russell, S.: Policy invariance under reward transformations: Theory and application to reward shaping. In: ICML, vol. 99, pp. 278–287 (1999)

Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT press, ??? (2018)

Woo, M., Cha, S.K., Gottlieb, S., Brumley, D.: Scheduling black-box mutational fuzzing. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS), pp. 511–522 (2013)

Elahi, H., Wang, G.: Forward-porting and its limitations in fuzzer evaluation. Information Sciences, 120142 (2024) https://doi.org/10.1016/j.ins.2024.120142

Schiller, N., Xu, X., Bernhard, L., Bars, N., Schloegel, M., Holz, T.: Novelty not found: Adaptive fuzzer restarts to improve input space coverage (registered report). In: Proceedings of the 2nd International Fuzzing Workshop, pp. 12–20 (2023)

Österlund, S., Razavi, K., Bos, H., Giuffrida, C.: {ParmeSan}: Sanitizer-guided greybox fuzzing. In: Proceedings of the 29th USENIX Security Symposium, pp. 2289–2306 (2020)

Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In: NDSS (2016)

Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T.: QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In: Proceedings of the 27th USENIX Security Symposium, pp. 745–761. USENIX, Baltimore (2018)

Österlund, S., Geretto, E., Jemmett, A., Güler, E., Görz, P., Holz, T., Giuffrida, C., Bos, H.: CollabFuzz: A Framework for Collaborative Fuzzing. In: Proceedings of the

14th European Workshop on Systems (EuroSec), pp. 1–7 (2021). https://doi.org/10.1145/3447852.3458720

Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., Hinton, H.: Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the USENIX Security Symposium, vol. 98, pp. 63–78 (1998). San Antonio, TX

Zheng, H., Zhang, J., Huang, Y., Ren, Z., Wang, H., Cao, C., Zhang, Y., Toffalini, F., Payer, M.: {FISHFUZZ}: Catch deeper bugs by throwing larger nets. In: Proceedings of the 32nd USENIX Security Symposium, pp. 1343–1360 (2023)

Scott, J., Sudula, T., Rehman, H., Mora, F., Ganesh, V.: Banditfuzz: fuzzing smt solvers with multi-agent reinforcement learning. In: International Symposium on Formal Methods, pp. 103–121 (2021). Springer

Hu, J., Duan, Y., Yin, H.: Marco: A stochastic asynchronous concolic explorer. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, pp. 1–12 (2024)