

# BandFuzz: An ML-powered Collaborative Fuzzing Framework

Wenxuan Shi

Northwestern University

wenxuan.shi@northwestern.edu

Hongwei Li

UC Santa Barbara

hongwei@ucsb.edu

Jiahao Yu

Northwestern University

jiahao.yu@northwestern.edu

Xinqian Sun

Northwestern University

xinqiansun2027@u.northwestern.edu

Wenbo Guo

UC Santa Barbara

henrygwb@ucsb.edu

Xinyu Xing

Northwestern University

xinyu.xing@northwestern.edu

**Abstract**—Collaborative fuzzing has recently emerged as a technique that combines multiple individual fuzzers and dynamically chooses the appropriate combinations suited for different programs. Unlike individual fuzzers, which rely on specific assumptions to maintain their effectiveness, collaborative fuzzing relaxes the assumptions on target programs, providing constant and robust performance across various programs. Ideally, collaborative fuzzing should be a more promising direction toward generic fuzzing solutions, as it mitigates the need for manual cherry-picking of individual fuzzers. However, the effectiveness of existing collaborative fuzzing frameworks is limited by major challenges, such as the need for additional computational resources compared to individual fuzzers and the inefficient allocation of resources among the various fuzzers.

To tackle these challenges, we present **BANDFUZZ**, an ML-powered collaborative fuzzing framework that outperforms individual fuzzers without requiring additional computational resources. The key technical contribution of **BANDFUZZ** lies in its novel resource allocation algorithm driven by our proposed multi-armed bandits model. Different from the greedy methods employed in existing collaborative fuzzing frameworks, **BANDFUZZ** models the long-term impact of individual fuzzers, enabling the discovery of globally optimal collaborative strategies. Moreover, we propose a novel fuzzer evaluation method that accesses not only code coverage but also the fuzzer’s capability of solving difficult branches. Finally, we integrate a real-time seed synchronization mechanism, as well as a set of implementation-wise optimizations to improve fuzzing efficiency and stability. Through extensive experiments on Fuzzbench and Fuzzer Test Suite, we show that **BANDFUZZ** outperforms state-of-the-art collaborative fuzzing framework, autofz, and widely used individual fuzzers. We also verify **BANDFUZZ**’s key designs through a comprehensive ablation study. Notably, we show **BANDFUZZ**’s effectiveness in real-world bug detection by analyzing the results of a worldwide fuzzing competition, where **BANDFUZZ** won the first place.

## 1. Introduction

The main purpose of fuzz testing is to thoroughly explore a program’s state space, uncovering potential vulnerabilities hidden in the program states. A comprehensive fuzzing test ensures the program is carefully tested with a wide variety of inputs, seeking to uncover as many vulnerabilities as possible. In recent years, there has been a significant research emphasis on expanding the program states that fuzzing can cover through optimizing key fuzzing components. These improvements include better techniques for selecting seeds [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], more efficient power scheduling [12], [13], and more effective ways to mutate the inputs [14], [15], [16], [17], [18], [19], [20], [21], [22]. All of these help fuzz testing cover more code branches for the program under test.

However, recent studies have shown progress in fuzzing technologies is slowing down [23]. The latest public benchmarks [24] and results from open competitions [25] support this. They show a lack of new academic fuzzers that can outperform well-established industry tools like AFL++ [26] from 2020. Reports from FuzzBench [27] and contests like 2023 SBFT Fuzzing Competition [28] also highlight that few fuzzers consistently perform better than others across different test targets. This shortfall mainly comes from the reliance on specific heuristics and assumptions in the design of research fuzzers. These assumptions may not match up well with the complexities seen in real-world programs, resulting in significant disparities in fuzzing performance.

Recognizing the limitations of individual fuzzing tools, recent research has started looking into collaborative fuzzing. This approach combines the strengths of multiple fuzzers into one unified fuzzer. The reasoning behind collaborative fuzzing is straightforward yet compelling: different fuzzers are capable of exploring different programs and program states. By combining the unique capabilities of various fuzzers, it is more likely to explore more comprehensive program states, achieving better coverage across diverse programs.

Recent studies have shown that collaborative fuzzers often outperform individual ones (e.g., [29], [30], [31], [32]). However, their effectiveness greatly relies on using a lot more additional computational resources compared to individual fuzzers. For example, recently collaborative fuzzing tools like EnFuzz [31], CollabFuzz [29], and Cupid [30] all need to run multiple fuzzers in parallel at the same time. This greatly hinders the usage of collaborative fuzzers, especially when fuzzing is done on systems with limited computational resources. As such, it is critical for collaborative fuzzers to efficiently *allocate resources* among the individual fuzzers they employ. This involves addressing the following three main challenges. First, to establish an effective way for the fuzzers to share knowledge. In a collaborative setting, different fuzzers may repeatedly explore the same code branches. Without a mechanism for the fuzzers to exchange code coverage information, these overlapping efforts could waste a lot of computing resources, undermining the overall fuzzing efficiency. Second, to develop a real-time fuzzer evaluation method that can accurately and dynamically evaluate the performance of each fuzzer during the fuzzing process. Effective resource allocation depends on identifying and prioritizing the most effective fuzzers. Without an accurate and real-time fuzzer evaluation, determining which fuzzers to assign more resources becomes difficult. Third, to implement a globally optimal resource allocation strategy that prioritizes not only the current effectiveness of individual fuzzers, but more importantly, their long-term impact. As a counter-example, we show that the greedy approach utilized by a recent collaborative fuzzer autofz [32] is sub-optimal, resulting in limited performance and potentially increasing the waste of computational resources.

To tackle these challenges, we introduce BANDFUZZ, an innovative collaborative fuzzing framework. Our key technical novelty is a multi-arm bandits-powered resource allocation method. Unlike greedy methods that allocate computational resources solely according to the immediate performance of individual fuzzers, our bandits-based approach balances fuzzers’ long-term effectiveness with their current performance, resulting in a more globally optimal resource allocation strategy. Furthermore, benefiting from the parametric efficiency of multi-arm bandits, our resource allocation method introduces only minimal computational overhead and does not require large-scale samples to learn an effective strategy. As explained in §3.3, standard bandit algorithms cannot be directly applied to our problem. As such, we introduce customized designs to a commonly used bandit algorithm, Thompson sampling, to make it suitable for our specific needs.

Our framework also integrates an innovative method for evaluating fuzzers. Our approach goes beyond the simple branch coverage metric and considers the effort needed to explore new branches. Specifically, we introduce a new metric to quantify the difficulty of exploring new branches and integrate it with coverage to form our final fuzzer evaluation metric. As shown in §5, our proposed metric provides a better assessment of a fuzzer’s real-world performance. Furthermore, we develop a real-time seed synchronization

mechanism to prevent individual fuzzers from redundantly exploring the same branches. We also conduct non-trivial implementation optimizations to enable real-time fuzzer evaluation and enhance the stability of our framework.

Through extensive evaluation on various fuzzing targets from two benchmarks, FuzzBench [33] and FTS [34], we first show that BANDFUZZ outperforms ten popular fuzzers (e.g., AFL++ [26], AFL [35], Angora [36], Darwin [16], MOPT [17] ) integrated in our framework. Second, we demonstrate that BANDFUZZ significantly outperforms the state-of-the-art collaborative fuzzing technique, autofz [32], confirming the superiority of our bandits-based resource allocations over greedy methods. Third, we conduct a comprehensive ablation study to verify the necessity of our three key components: seed synchronization, fuzzer evaluation, and bandits-based resource allocation. Fourth, we run a sensitivity test to confirm the effectiveness of our customized bandits model and the in-sensitivity to key hyper-parameters. More importantly, we used BANDFUZZ to participate in a recent world-level fuzzing competition. BANDFUZZ won the *first place*, beating state-of-the-art research and industrial-level fuzzers. The competition also highlights BANDFUZZ’s superior capability in vulnerability detection. To the best of our knowledge, BANDFUZZ is the first ML-powered collaborative fuzzing framework that demonstrates its effectiveness and stability across a wide range of programs.

In summary, we make the following contributions.

- We introduce BANDFUZZ, an ML-powered collaborative fuzzing framework. BANDFUZZ integrates an accurate and real-time fuzzer evaluation method, a real-time seed synchronization method, and a novel mechanism that uses our customized bandits algorithm to allocate resources.
- We implement BANDFUZZ to incorporate ten widely-used fuzzers, demonstrating its superiority over state-of-the-art individual and collaborative fuzzing tools.
- We comprehensively evaluate the necessity of three main designs in BANDFUZZ and its insensitivity to key hyper-parameters. We further provide an analysis of a recent fuzzing competition result, where BANDFUZZ beats cutting-edge research fuzzers and industrial-level fuzzers.

## 2. Background

A wide range of research has been conducted on program fuzzing, encompassing everything from the development of individual fuzzers to the implementation of collaborative fuzzing that combines multiple fuzzers. In the following, we briefly summarize these research endeavors, with a primary focus on recently emerged collaborative fuzzing frameworks and their limitations.

### 2.1. Individual Fuzzers

Existing research on individual fuzzer development mainly focuses on improving the following components: seed selection, power scheduling, mutation strategies, and integration of complex analysis methods.

**Seed selection.** Beyond standard seed selection strategies that consider execution efficiency and seed length, research in this direction has explored other factors [4], [5], [6], [7], [8], [9], [10], [11], [12], [14], [37], [2]. For example, NEUZZ [1] considers branch uniqueness, Cerebro [3] leverages code complexity, and EcoFuzz [13] uses state transition probability.

**Power scheduling.** Another line of research focuses on improving power scheduling [12], [13], [8], [38], [39], [40]. For example, AFLFast [12] leverages a Markov chain model to allocate more energy to seeds that reach paths that are infrequently visited.

**Mutation strategies.** This research area focuses on creating new mutation techniques or mutator scheduling strategies [14], [15], [16], [17], [18], [19], [20], [21], [22]. Notably, FairFuzz [15] prioritize the mutation of input bytes linked to rarely executed branches. RedQueen [18] develops another method for selecting important input bytes to prioritize mutations. DARWIN [16] and MOpt [17] design a novel mutator scheduling strategy based on the evolution strategy and Particle Swarm Optimization (PSO), respectively.

**Integration of concolic execution and taint analysis.** Some fuzzers incorporate concolic execution [41], [42], [43], [44], [45], [46], [47], [48] or taint analysis [36], [49], [50], [51] to better tackle difficult branches and reduce fuzzing search space, enhancing the overall fuzzing efficiency. Some representative fuzzers include Angora [36], SymCC [43], and SymSan [42], which leverage these advanced techniques to explore challenging code paths more effectively.

**Limitations.** Although individual fuzzers exhibit state-of-the-art performance in specific contexts, their effectiveness may decrease under different conditions, resulting in large variations in fuzzing performances across different programs. This variation is due to the specific assumptions that each fuzzer relies on, which may not apply to all programs. Consequently, no single fuzzer emerges as a universally optimal solution, capable of delivering consistently superior performance across a wide range of programs.

## 2.2. Collaborative Fuzzing

To tackle the limitation of individual fuzzers, recent research is shifting focus to collaborative fuzzing that integrates the strengths of multiple fuzzers to enhance the fuzzing effectiveness and stability across diverse programs.

**EnFuzz** [31] integrates six fuzzers, which are handpicked by domain experts. EnFuzz runs the selected fuzzers in parallel, each taking a separate CPU core. It also uses a dedicated thread to synchronize seeds every two minutes (i.e., get a union seed set and copy it to every fuzzer), ensuring a shared and up-to-date seed set for all participants.

**CollabFuzz** [29] retains EnFuzz’s architecture and fuzzers but extends its capabilities with advanced seed synchronization policies, i.e., providing more options for what seeds to synchronize (e.g., prioritizing important seeds) and when to synchronize (e.g., immediately upon a seed is generated).

| Code                                                           | AFL | AFL++ |
|----------------------------------------------------------------|-----|-------|
| if (strcmp(dev, "dbus-system") == 0) {                         | •   | •     |
| if (!(handle->conn = dbus_bus_get(DBUS_BUS_SYSTEM, &error))) { |     | •     |
| return PCAP_ERROR; }                                           |     | •     |
| } else if (strcmp(dev, "dbus-session") == 0) {                 | •   |       |
| // ...                                                         |     |       |
| } else if (strcmp(dev, "dbus://" , 7) == 0) {                  | •   |       |
| // ...                                                         |     |       |
| } else {                                                       | •   |       |
| snprintf(handle->errbuf, ...);                                 | •   |       |
| return PCAP_ERROR; }                                           | •   |       |
| // main program logic starts below ...                         |     |       |

Figure 1: An example of two individual fuzzer’s code coverage on libpcap. The dense cycle points out the resolved branches.

**Cupid** [30] first conducts an offline evaluation of eight fuzzers on some pre-selected programs and records the probability for each fuzzer to cover each branch. It requires the user to decide the number of fuzzers to use. Then, Cupid selects the optimal fuzzer combination that maximizes the average branch coverage. Like EnFuzz and CollabFuzz, each fuzzer runs on a separate CPU core, with seed synchronization akin to EnFuzz’s approach.

**autofz** [32] is different from the approaches of EnFuzz, CollabFuzz, and Cupid. It is tailored to perform fuzzing under resource constraints. Specifically, it proposes a straightforward exploration-exploit method for resource allocation among the fuzzers it integrates. During the exploration phase, autofz allocates the same resource to each fuzzer and evaluates them by unique coverage. Based on these evaluations, the subsequent exploitation phase allocates more resources to fuzzers that demonstrated superior performance during the exploration phase. This allocation strategy essentially employs a greedy algorithm that uses locally optimal solutions in each fuzzing period to approximate a globally optimal solution in the entire fuzzing campaign.

**Limitations.** Despite the novel designs, existing collaborative fuzzing techniques have not achieved the same level of efficacy as widely used individual fuzzers. Below, we outline the reasons for this discrepancy.

- **Requiring additional computational resources.** EnFuzz, CollabFuzz, and Cupid run selected fuzzers in parallel, which means they all necessitate more computational resources than individual fuzzers. Consequently, enhancements in fuzzing performance can be attributed to the increased resources used. This design limits their effectiveness when resources are constrained.
- **Infrequent cross-fuzzer knowledge/seed synchronization.** During fuzz testing, fuzzers included in autofz and other collaborative fuzzing frameworks either do not share their path coverage or do so infrequently. This results in a considerable waste of resources, as areas of the program space left unexplored by one fuzzer might already have been covered by another.
- **Non-real-time and inaccurate fuzzer evaluation.** Existing collaborative fuzzers are inefficient in measuring and tracking the real-time effectiveness of individual fuzzers throughout the fuzzing procedure. Specifically, EnFuzz and CollabFuzz directly run the selected fuzzers with-

out any evaluation. Cupid involves an offline evaluation, where the measurement of individual fuzzers remains unchanged throughout the fuzzing process. autofz assesses each fuzzer based on the uniqueness of code coverage it achieves during its exploration phase. Specifically, fuzzers that uncover more unique paths are rated higher. However, this approach may limit the fuzzing exploration to only local areas of the codebase. For instance, as illustrated in Figure 1, AFL might achieve a higher unique branch coverage than AFL++ but is limited to the input format validation phase, failing to reach the deeper and more critical part of the main program logic. In contrast, AFL++ is capable of identifying critical branches with complex conditions (Line 2), enabling subsequent fuzzing processes to reach the main program logic and potentially achieve a higher branch coverage at a later stage. Furthermore, autofz’s fuzzing evaluation is not real-time.

- **Inefficiencies in resource allocation strategy.** EnFuzz, CollabFuzz, and Cupid allocate a CPU core to each fuzzer without any resource allocations and thus cannot adapt to performance changes in individual fuzzers. autofz explicitly separates the fuzzing process into multiple sub-periods and formulates a fixed resource allocation strategy for each period. This greedy approach does not consider potential relationships between different periods, lacking a global assessment. As demonstrated in existing works [17], [52], [53], [54] and our evaluation in §5, a decision-making process without a global perspective results in a strategy that is not globally optimal.

### 3. Key Techniques

To address the limitations above, we design and develop BANDFUZZ. At a high level, we first propose a novel evaluation method for real-time and accurate measurement of dynamic performance changes of individual fuzzers. More importantly, we design a real-time resource allocation mechanism utilizing multi-arm bandits, which accounts for both local and global assessments. In this section, we first provide an overview of our design rationale and then delve into the technical details.

#### 3.1. Overview of BANDFUZZ

**Real-time and faithful fuzzing evaluation.** As discussed in §2.2, enabling real-time fuzzer evaluation is crucial to optimize resource allocation for more effective fuzzers. To achieve this, a straightforward solution is to use each fuzzer’s coverage at each round as the metric. As detailed in §4, we conduct non-trivial implementation efforts to efficiently return the coverage of individual fuzzers at each fuzzing round. However, coverage is not a proper metric. First, different fuzzers may exhibit the same coverage, making it difficult to prioritize fuzzers with the same coverage. Second, the difficulty and importance of identifying different branches varies a lot. As shown in Figure 1, a branch (Line 2) covered by AFL++ with a complex condition is more important than other branches, as it enables the

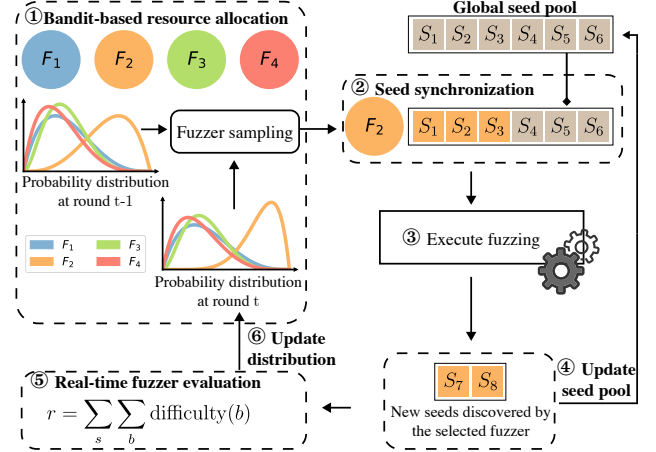


Figure 2: The overview of BANDFUZZ. In the figure, BANDFUZZ manages four fuzzers. In the current round  $t$ , the fuzzer  $F_2$  is chosen. Initially,  $F_2$ ’s seed pool contains seeds  $S_1 \sim S_3$ , while the global seed pool contains  $S_1 \sim S_6$ . After seed synchronization,  $F_2$  extends its seed pool with the missing seeds and then proceeds to execute the fuzzing task, discovering new seeds  $S_7$  and  $S_8$ . The new seeds are added to the global seed pool and evaluated to get a reward to update the probability distribution for  $F_2$ .

exploration of main program logic. Fuzzers unlocking more crucial branches should be deemed more effective than those addressing simpler ones. This evaluation cannot be achieved solely with coverage information.

To address these limitations, autofz treats unique paths as difficult ones. As discussed in §2.2, this metric may limit exploration to local regions. Furthermore, it is computationally inefficient, making it difficult to conduct real-time evaluation of fuzzing dynamics. In this work, we propose a more efficient metric for measuring the branch difficulty (detailed in §3.2). At a high level, we employ the time required to discover each branch as the measure of its difficulty. If a branch takes longer to be found, we consider it more challenging. Unlike branch uniqueness, recording branch discovery time introduces negligible overhead, significantly enhancing efficiency. We integrate this metric with branch coverage and use the combined metric (denoted as *fuzzer evaluation metric*) to evaluate the effectiveness of individual fuzzers in each fuzzing round.

**Multi-arm bandits based resource allocation.** After enabling real-time evaluation of fuzzing dynamics, the next key step for effective collaborative fuzzing is to optimize resource allocation based on the effectiveness of individual fuzzers. Here, we discuss the single-core setup, and the goal is to assign more resources to more effective fuzzers throughout the fuzzing process. Straightforward solutions involve either consistently selecting the best fuzzer after each evaluation or allocating a specific amount of resources to each fuzzer based on their effectiveness ranking, as employed in autofz. As discussed in §2.2, such greedy solutions lack the flexibility to explore allocation strategies that may not be optimal in the current period but could lead to better performance in longer runs. As a result, such solutions fall

short of achieving globally optimal performance throughout the entire fuzzing campaign. Essentially, this is the problem of balancing exploitation and exploration.

We propose to leverage the multi-arm bandit to solve this problem. Multi-arm bandits, or simply *bandits*, represent a set of algorithms that learn efficient strategies for searching and scheduling problems [55], [56], [57], [58], [59]. Their objective is to strike a balance between exploiting the current optimal choices (exploitation) and exploring potentially better choices (exploration). Bandits naturally align with our problem of balancing exploitation and exploration in resource allocation. Furthermore, in contrast to other learning problems, bandits do not require pre-collection of a dataset or training a substantial number of parameters. Their real-time and efficient nature further sets them as a natural fit for our problem. As we will detail in §3.3, we build a customized bandit model for our problem, which treats individual fuzzers as an arm and our proposed fuzzer evaluation metric as the reward. We leverage the Thompson sampling algorithm to determine the weights for individual fuzzers in each fuzzing round and dynamically allocate resources in real time based on these weights.

**Overall procedure of BANDFUZZ.** As demonstrated in Figure 2, in each round, given a weight distribution of individual fuzzers, BANDFUZZ first sample a fuzzer  $F_2$  based on this distribution (i.e.,  $F_2$  is selected in round  $t$ ) (①). It then conducts a seed synchronization (②), updating the seed pool of  $F_2$  by incorporating the global seed pool (i.e., the union set of seeds from all fuzzers). BANDFUZZ then runs  $F_2$  for a certain cycle, in which each cycle performs the seed selection, energy assignment, and mutations of  $F_2$  (③). After fuzzing with  $F_2$ , BANDFUZZ updates the global seed pool with newly discovered seeds (④) and recomputes the reward of individual fuzzers using our proposed fuzzer evaluation metric (⑤). Finally, BANDFUZZ updates the weight distribution based on the reward using our customized bandit model (Thompson sampling) (⑥). BANDFUZZ continuously performs this iterative process until a predefined stopping criterion is reached.

Note that our design also addresses the first two limitations highlighted in §2.2. First, BANDFUZZ distributes the given resources among the integrated fuzzers without needing extra resources. Second, as detailed in Section 4, BANDFUZZ incorporates a global seed pool to enable real-time coverage sharing.

### 3.2. Fuzzer Evaluation Methodology

As described above, we introduce a novel fuzzer evaluation approach designed to gauge the performance of fuzzers and subsequently allocate rewards accordingly. In our framework, a high reward signifies a fuzzer’s effectiveness. Below, we detail our evaluation methodology and the associated reward scheme.

To evaluate a fuzzer’s performance, we consider not only its capability to uncover new branches but also the effort required to achieve such discoveries. This consideration stems from an observation that, during the initial stages of

---

#### Algorithm 1: Assign reward to a selected fuzzer

---

**Input:** The current fuzzing round  $t$ , the selected fuzzer  $F_t$ , the set of new seeds generated by this fuzzer  $S_t$ , the maximum and minimum reward  $r_{\max}$  and  $r_{\min}$ , and the set  $\mathcal{I}$ , recording the round at which each basic block is *firstly* discovered.

```

1  $r_t \leftarrow 0$ ;
2 for seed  $S$  in  $S_t$  do
3   for new branch  $b$  covered by  $S$  do
4     Suppose the branch  $b$  goes from the basic
       block  $BB_p$  to  $BB_e$ ;
5     Find the round that  $BB_p$  is firstly
       discovered in  $\mathcal{I}$ , noted as  $t_p$ ; Compute the
       coverage interval  $c \leftarrow t - t_p$ ;
6     Update the reward  $r_t \leftarrow r_t + c$ ;
7     if  $BB_e$  is newly discovered then
8       | Add the pair  $BB_e$  and  $t$  to  $\mathcal{I}$ ;
9     end
10  end
11 end
12  $r_{\max} \leftarrow \max(r_{\max}, r_t)$ ;
13  $r_{\min} \leftarrow \min(r_{\min}, r_t)$ ;
14  $r_t \leftarrow (r - r_{\min}) / (r_{\max} - r_{\min})$ ;
Output: normalized reward  $r_t$ 
```

---

fuzzing, individual fuzzers generate a large number of seeds, contributing to an increase in coverage. However, as time progresses, identifying new branches becomes increasingly difficult, often leading to a stagnation in coverage growth. At this point, while most fuzzers face challenges in discovering new branches despite numerous mutation attempts, a select few manage to overcome complex constraints, generating seeds that unlock critical branches (as shown in Figure 1). This breakthrough leads to a rapid emergence of new seeds, facilitated by the resolution of complex constraints and the subsequent ease of discovering additional branches.

Following this observation, we design our fuzzer assessment mechanism and reward scheme as follows. After running the selected fuzzer  $F_t$  at the fuzzing round  $t$ , we obtain a set of new seeds, represented as  $S_t$ . The reward of the selected fuzzer, denoted as  $r_t$ , is equal to the accumulated rewards of all seeds  $S \in S_t$ . Each seed’s reward is subsequently calculated from the accumulated rewards of all unique and newly covered branches ( $b$ ) that are associated with it. As indicated from Line 4 to Line 6 in Algorithm 1, each branch’s reward corresponds to its coverage interval ( $c$ ). The latter quantifies how many rounds were required to discover this new branch. Mathematically speaking, coverage interval  $c$  equals the difference between the current round number ( $t$ ) and when its preceding block (denoted as  $BB_p$ ) was *firstly* discovered. We calculate  $r_t$  by summing up values of  $c$  over all newly covered branches across all seeds in  $S_t$ . Finally, we normalize  $r_t$ , using min-max scaling, to determine the final reward for fuzzer  $F_t$  (Line 12-14). This approach allows us not only to capture overall coverage

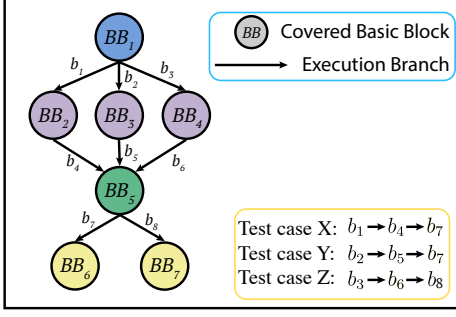


Figure 3: Motivating example for demonstrating reward computing.

through the accumulation of newly discovered branches but also to account for the effort required via the computation of individual branches' coverage interval  $c$ .

**Example of reward computing.** As demonstrated in Figure 3, the program features a target program with eight distinct branches. Initially, seed  $X$  identifies branches  $b_1$ ,  $b_4$ , and  $b_7$  during iteration  $I_X$ , leading to the inclusion of  $\{BB_1 = I_X, BB_2 = I_X, BB_5 = I_X, BB_6 = I_X\}$  in set  $I$ . Following a series of iterations without new discoveries, seed  $Y$  emerges in round  $t = I_Y$ , detecting branches  $b_2$ ,  $b_5$ , and  $b_7$ , with  $b_2$  and  $b_5$  being newly identified. For branch  $b_2$ ,  $BB_1$  is recognized as the preceding block ( $BB_p$ ) and  $BB_3$  as the subsequent block ( $BB_e$ ), resulting in a coverage interval calculation of  $c = t - t_p = I_Y - I_X$ . Consequently,  $\{BB_3 = I_Y\}$  is appended to set  $I$ . Regarding  $b_5$ ,  $BB_3$  is the preceding block and  $BB_5$  is the subsequent block, leading to a coverage interval of  $c = I_Y - I_Y = 0$ . Since  $BB_5$  was previously discovered, set  $I$  remains unchanged. The fuzzer that generates seed  $Y$  is then awarded a reward  $r_t = (I_Y - I_X) + (0) = I_Y - I_X$ . After additional rounds without findings, seed  $Z$  is identified at round  $t = I_Z$ , uncovering new branches  $b_3$ ,  $b_6$ , and  $b_8$ . The coverage interval for  $b_3$  is calculated as  $c = I_Z - I_X$ , with  $\{BB_4 = I_Z\}$  added to set  $I$ . The interval for  $b_6$  is  $c = I_Z - I_Z = 0$ , and for  $b_8$ , it is  $c = I_Z - I_X$ , leading to the inclusion of  $\{BB_7 = I_Z\}$  in set  $I$ . Ultimately, the fuzzer that generates seed  $Z$  is awarded a reward  $r_t = (I_Z - I_Y) + (0) + (I_Z - I_X) = 2I_Z - I_Y - I_X$ .

### 3.3. Resource Allocation Mechanism

Our resource allocation algorithm is shown in Algorithm 2. After initializing the required variables (Lines 1-6), in each fuzzing round, our first step is to select a fuzzer  $F_t$  using our customized bandits model (Line 8), followed by synchronizing its seeds with those in the global seed pool (Line 9). Subsequently, we execute  $F_t$ , gather its reward, and update both its probability distribution and the global seed pool (Lines 10-16). We also employ the auto-cycle mechanism to revise how many cycles a fuzzer should run when it is selected next time (Lines 17-20). Finally, after updating the model for certain rounds, we reset parameters within our bandits model to ensure optimal performance moving forward (Lines 21-24). In the following, we introduce our customized bandits model and the auto-cycle method.

#### Algorithm 2: BANDFUZZ resource allocation

**Input:** A set of individual fuzzer  $\mathcal{F}$ , global seed pool  $\mathcal{S}_g$ , reset interval  $I_R$ , time budget for each round  $T_I$ , stopping condition

```

1 timer  $\leftarrow 0$ , Number of round  $t \leftarrow 0$ ;
2 for fuzzer  $F$  in  $\mathcal{F}$  do
3    $F.total\_duration \leftarrow 0$ ,  $F.num\_selection \leftarrow 0$ ;
4    $F.cycles \leftarrow 1$ ;
5    $\alpha_F \leftarrow 1$ ,  $\beta_F \leftarrow 1$ ;
6 end
7 while stopping condition is not met do
8   Current fuzzer  $F_t \leftarrow TS(\mathcal{F})$  (①);
9   Synchronize seeds from  $\mathcal{S}_g$  to the seed pool of  $F_t$  (②);
10   $start \leftarrow$  current time;
11  for  $j=0, j \leq F_t.cycles, j++$  do
12     $F_t.run\_one\_cycle()$  (③);
13  end
14  Update  $\mathcal{S}_g$  with  $\mathcal{S}_t$ , new seeds generated by  $F_t$  (④);
15   $d_t \leftarrow$  current time  $- start$ ,  $r_t \leftarrow$  evaluate  $F_t$  with Algorithm 1 (⑤);
16  // Update probability distribution (⑥)
17   $\alpha_{F_t} \leftarrow \alpha_{F_t} + \hat{r}_t$ ,  $\beta_{F_t} \leftarrow \beta_{F_t} + (1 - \hat{r}_t)$ ;
18  // Auto-cycle
19   $F_t.total\_duration \leftarrow F_t.total\_duration + d_t$ ;
20   $F_t.num\_selection \leftarrow F_t.num\_selection + 1$ ;
21   $F_t.avg\_cycle\_time \leftarrow \text{floor}(\frac{F_t.total\_duration}{F_t.num\_selection})$ ;
22   $F_t.cycles \leftarrow \frac{T_I}{F_t.avg\_cycle\_time}$ ;
23  // Reset mechanism
24   $timer \leftarrow timer + d_t$ ,  $t \leftarrow t + 1$ ;
25  if  $timer \geq I_R$  then
26     $timer \leftarrow 0$ ,  $\alpha_F \leftarrow 1$ ,  $\beta_F \leftarrow 1$  for every fuzzer;
27  end
28 end

```

**Customized bandits model.** Each arm represents an individual fuzzer  $F \in \mathcal{F}$ , associated with a weight distribution. Our goal is to keep updating the weight distributions in a way that maximizes the long-term accumulated reward obtained through repeated pulls of the arms. By maximizing the long-term reward, the bandits can select the fuzzer benefiting the overall fuzzing campaign. In contrast, the greedy method selects the fuzzers with the highest reward at each time but may not result in an optimal long-term reward.

We use the Thompson sampling algorithm (TS) [57] to update the weight distributions. TS has both the SOTA empirical performance and theoretical guarantee in various problems [56], [60]. Suppose the fuzzer set  $\mathcal{F}$  contains  $K$  fuzzers. For each fuzzer (or arm)  $F_k$ , TS defines the conditional distribution of the reward as a Bernoulli distri-

---

**Algorithm 3:** Customized TS bandit algorithm

---

**Input:**

```
1  $r_t \leftarrow 0$ ;  
2 for  $t = 1, 2, \dots$  do  
    // sample model  
3   for  $k = 1$  to  $K$  do  
4     Sample  $\hat{\theta}_k \sim \text{Beta}(\alpha_k, \beta_k)$   
5   end  
    // select and apply action  
6    $x_t \leftarrow \arg \max_k \hat{\theta}_k$   
7   Apply  $x_t$  and observe  $r_t$   
8   Sample  $\hat{r}_t \sim \text{Bern}(r_t)$   
    // update distribution  
9    $(\alpha_{x_t}, \beta_{x_t}) \leftarrow (\alpha_{x_t} + \hat{r}_t, \beta_{x_t} + 1 - \hat{r}_t)$   
10 end
```

---

bution parameterized by  $\theta_{F_k}$ . Recall that in §3.2 we use  $r_t$  to denote the reward of  $F_t$ . So  $r_t|F_k \sim \text{Bern}(\theta_{F_k})$ . TS then defines the distribution of  $\theta_{F_k}$  as a Beta distribution, parameterized by  $\alpha_{F_k}$  and  $\beta_{F_k}$ , i.e.,  $\theta_{F_k} \sim \text{Beta}(\alpha_{F_k}, \beta_{F_k})$ . This is the weight distribution for  $F_k$ . In each fuzzing round  $t$ , TS first samples  $\theta_{F_k}$  from the current weight distribution for each fuzzer. Then, it selects the fuzzer with the largest  $\theta_{F_k}$ , i.e.,  $F_t = \arg \max_{F_k} \theta_{F_k}$ . After running the selected fuzzer and collecting the reward  $\hat{r}_t$ , TS then updates the parameters of the weight distribution of the selected fuzzer as follows:

$$\alpha_{F_t} \leftarrow \alpha_{F_t} + \hat{r}_t, \quad \beta_{F_k} \leftarrow \beta_{F_k} + 1 - \hat{r}_t. \quad (1)$$

Directly applying the original TS introduces two challenges. First, TS assumes  $\hat{r}_t \in \{0, 1\}$  is discrete, while our reward falls in a continuous range of  $[0, 1]$ . To overcome this discrepancy, we employ a method in [61] to discretize our continuous reward  $r_t$  into a binary equivalent  $\hat{r}_t$ . This discretized reward is then used to update the parameters following Equation 1, aligning our implementation with the TS framework.

TS also assumes that the reward distributions remain stationary over time. However, in our scenario, as more program branches are discovered over time, these reward distributions will change accordingly. To account for such variations, we periodically reset each fuzzer’s weight distribution parameters. This approach provides flexibility in capturing changes in the evolving reward distributions (see §A for more details).

**Auto-cycle mechanism.** After choosing a fuzzer in each round, it is also necessary to determine the time and resources allocated to it. The guiding principle is fairness; we aim to assign similar resources in each round. It aligns with the bandits’ assumption that rewards are obtained under similar resource conditions in every round. A straightforward approach is to allocate a fixed run-time for each round. Unfortunately, this will break the fuzzing procedure of individual fuzzers. Instead, we specify the number of fuzzing cycles to run (one cycle means the full procedure from seed selection to generating new seeds), which better maintains

the fuzzing procedure. Assigning the same number of cycles is also not an ideal approach, as some fuzzers operate faster than others, leading to unfairness. To address this issue, we introduce an auto-cycle mechanism that dynamically adjusts the number of fuzzing cycles for each fuzzer based on its previous execution time. Specifically, for each fuzzer, we first calculate its average cycle time ( $F.avg\_cycle\_time$ ). We then establish a fixed time budget  $T_I$ , which determines how long a fuzzer can run in any given round. The number of cycles is computed as  $\frac{T_I}{F.avg\_cycle\_time}$ .

## 4. Implementation

In this section, we provide a comprehensive overview of the implementation, focusing on the integration, management, synchronization, and evaluation of fuzzers.

### 4.1. Fuzzer Integration

BANDFUZZ currently supports 10 individual fuzzers, namely: AFL [35], AFL++ [26], AFLFast [12], Angora [36], Darwin [16], FairFuzz [15], honggfuzz [62], LAF-Intel [63], MOPT [17] and Radamsa [19]. These fuzzers were selected because of their status as state-of-the-art tools that incorporate diverse strategies to enhance various components of fuzzing. These aforementioned fuzzers operate on four different frameworks. Specifically, the AFL [35] framework underpins the operation of AFL, AFLFast, Darwin, and FairFuzz. Meanwhile, AFL++, LAF-Intel, MOPT, Radamsa are based on the AFL++ framework [26]. Angora and honggfuzz each utilize their own unique frameworks. Note that we enable `cmplog` within AFL++, a technique analogous to that utilized in RedQueen. `cmplog` is integrated as the default mode for AFL++.

In order to integrate these disparate fuzzers into a unified runtime environment, necessary but minimal modifications have been made to their underlying frameworks. For example, fewer than 200 lines of code in both the AFL and AFL++ frameworks have been modified. It is important to note that these integrated fuzzing frameworks are among the most widely used in this field. They serve as the platforms for many fuzzers [13], [52], [44], [54], [3], [64], [65], [66]. This widespread usage suggests that BANDFUZZ can effortlessly integrate additional fuzzers with existing modifications.

### 4.2. Fuzzer Management, Synchronization & Evaluation

BANDFUZZ uses a central communication interface that functions as a bridge to connect individual fuzzers. This interface is responsible for implementing most functionalities of BANDFUZZ and coordinating with each fuzzer.

**Fuzzer management.** The central communication interface governs the operations of each fuzzer, including initiation, pausing, and resumption of their fuzzing tasks. It also monitors and reports on the runtime status of every fuzzer (e.g.,

determining when a specific fuzzer is engaged in seed selection). During their operational cycles, individual fuzzers may encounter two potential issues. The first issue arises when they become trapped in an infinite loop, resulting in an unexpectedly prolonged execution time for their current task. The second problem occurs if they fail and shut down during operation. These complications can suspend BANDFUZZ and negatively impact its overall performance. If the central interface detects that a currently active fuzzer has taken an unusually long time without completing its assigned cycles, it will dispatch a skip instruction to that particular fuzzer. This command pauses the fuzzer’s operation and returns the current results to the central interface. In instances where a fuzzer fails prematurely before completing its designated cycles, the central interface will send out a restart instruction to reboot it. Consequently, this allows for continuation toward the completion of its assigned fuzzing cycles. These dual mechanisms ensure all fuzzers adhere strictly to round-time constraints  $T_I$ , even under circumstances involving infinite loops or failures.

**Real-time synchronization.** Simultaneously, this interface receives discoveries (i.e., seeds) from all connected fuzzers. As previously detailed in §3.1, BANDFUZZ operates by maintaining a global seed pool. The first step when selecting a fuzzer involves synchronizing its local seed pool with the global one. This synchronization process is achieved by tracking the local seed files present in the file system and identifying any missing seeds. Given that the central interface has knowledge about each individual fuzzer’s status, it can efficiently transmit these missing seeds to the chosen fuzzer prior to any form of seed selection. Furthermore, we use this same method to monitor changes in the file system in order to enrich our global seed pool with new seeds. Consequently, newly discovered seeds are transferred from selected fuzzers into our global pool via this central interface.

**Real-time evaluation.** Once the central interface acknowledges the new seeds produced by the selected fuzzer, BANDFUZZ incorporates them into set  $\mathcal{I}$ , as delineated in Algorithm 1. Subsequently, it computes the reward for the selected fuzzer through an evaluation of these newly produced seeds. This computation is facilitated using a specially customized instrumentation binary equipped with coverage sanitizer and a customized runtime shared object that executes Algorithm 1.

## 5. Experiment

In this section, we comprehensively evaluate BANDFUZZ to answer the following research questions (RQs):

**RQ1.** BANDFUZZ integrates a suite of advanced fuzzing tools, leveraging their combined strengths. When provided with identical computational resources, how does the collaborative fuzzing approach of BANDFUZZ compare against the performance of the individual fuzzers it incorporates?

**RQ2.** When compared to the state-of-the-art collaborative fuzzing method, `autofz`, does BANDFUZZ, enhanced

with our proposed fuzzer evaluation and resource allocation method, demonstrate better performance?

**RQ3.** BANDFUZZ introduces three key techniques, including real-time synchronization via a global seed pool, fuzzer evaluation, and resource allocation. How do these key components individually contribute to the overall effectiveness of BANDFUZZ?

**RQ4.** To what extent does our customized bandits model improve the overall performance of BANDFUZZ?

**RQ5.** Whether BANDFUZZ is sensitive to the key hyperparameters (time budget for each round  $T_I$  and the reset interval  $I_R$ )?

In the following, we specify our experiment setup, as well as the design and results of the experiment to answer each research question.

### 5.1. Experiment Setup

**Experiment environment.** We conduct all experiments on two workstations, each equipped with an AMD EPYC 7763 64-core processor and 512GB of RAM. Within this setup, each fuzzer operates in a separate Docker container (note that BANDFUZZ is considered a single fuzzer). Each docker container (fuzzer) is allocated with a single CPU core and 2GB of shared memory, ensuring a standardized, controlled, and fair testing environment.

**Configuration of BANDFUZZ.** Recall that BANDFUZZ periodically reset the bandits parameters to adapt to changes in reward distributions. Also, BANDFUZZ requires specifying an appropriate time budget for the fuzzer selected by the bandits during the auto-cycle. The default setup for these two hyperparameters is that the reset interval  $I_R$  is 120 mins, and the time budget  $T_I$  is 120 seconds for each round.

**Benchmark and fuzzing targets.** We select the fuzzing targets in the benchmark database FuzzBench [33]. We also use the Fuzzer Test Suite (FTS) [34] as an alternative benchmark because some fuzzers are not compatible with the targets in FuzzBench. Each fuzzer, representing both our proposed solution and established fuzzing techniques, was subjected to a continuous testing period of 24 hours for every fuzzing target. To mitigate inherent variability and ensure robust results, we executed each experiment 10 times, resulting in a cumulative computation time of 7.5 CPU years.

**Metrics.** We select the widely used branch coverage as the our metric for evaluating the effectiveness of a fuzzer against a specific target program. Considering the varying performance that different fuzzers show across different programs, we calculate the average score for each fuzzer across different programs to enable a more comprehensive comparison. The average score is introduced by FuzzBench [33], which serves as a standard metric to evaluate the overall performance of a fuzzer. This metric allows for a more balanced comparison by accounting for the performance difference across multiple target programs, thereby offering a comprehensive view of a fuzzer’s effectiveness. More specifically, the fuzzer’s score for each target equals dividing its median branch coverage by the maximum branch coverage

recorded for that same target. Note that the highest relative performance does not necessarily correspond to 100%.

## 5.2. Experiment Design

**Experiment I: Comparison with individual fuzzers.** To address RQ1, we isolate the fuzzers incorporated within BANDFUZZ, treating them as individual fuzzers. We then run BANDFUZZ and each individual fuzzer against the targets from FuzzBench. As mentioned in §5.1, we run each fuzzer 24 hours with the same identical computational resources. We compute the branch coverage for each method in each individual target and the mean coverage across all targets. We run each method 10 times and report the mean. We expect that BANDFUZZ will have higher overall fuzzing performance compared to individual fuzzers, as BANDFUZZ’s can dynamically evaluate and select the most efficient fuzzers for different targets over time.

**Experiment II: Comparison with autofz.** To address RQ2, we compare BANDFUZZ with the state-of-the-art collaborative fuzzing framework autofz. It integrates 11 distinct fuzzers. In Experiment I, we use the fuzzing targets in FuzzBench. However, the implementation of autofz is incompatible with FuzzBench. As such, we use FTS as the benchmark for this experiment. Note that despite both autofz and BANDFUZZ incorporate multiple fuzzers, the composition of these fuzzers is not identical, featuring unique fuzzers such as Darwin in BANDFUZZ and libFuzzer in autofz. To enable an apple-to-apple comparison, we restrict the individual fuzzers to those mutually supported by both autofz and BANDFUZZ. Specifically, we craft autofz-8 and BANDFUZZ-8 that integrate the following eight fuzzers: AFL, AFL++, AFLFast, LAF-Intel, MOpt, Radamsa, Angora, Fairfuzz. We compare the branch coverage between autofz-8 and BANDFUZZ-8 to evaluate which method has the better collaboration strategy on the same set of fuzzers.

We also compared BANDFUZZ-8 with the complete setup of autofz, which includes all 11 fuzzers. This comparison aimed to evaluate the strengths of BANDFUZZ against a well-established benchmark in the field. Demonstrating that BANDFUZZ-8 outperforms the full autofz would further support the effectiveness of BANDFUZZ.

**Experiment III: Ablation study.** We conduct a comprehensive ablation study to answer RQ3 (i.e., the efficacy of our proposed seed synchronization, fuzzer evaluation, and resource allocation mechanism). First, we configure BANDFUZZ to operate without conducting seed synchronization with the global seed pool (denoted as “no-sync”). Comparing no-sync with BANDFUZZ can quantify the importance of our seed synchronization mechanism. Our hypothesis is that without a synchronization mechanism, there is a possibility for the fuzzers in BANDFUZZ to redundantly explore the same paths, leading to a waste of time and resources.

Second, to evaluate the importance of our fuzzer evaluation method, we adjust the reward allocation for fuzzer  $F$  in Algorithm 1 by incrementing  $\alpha_F$  by 1 upon any new coverage discovery and increasing  $\beta_F$  by 1 when it fails. Additionally, we deactivated the auto-cycle feature to

facilitate precise reward calculation. We denote this variation as “naive-reward”. Comparing this variation with the simple fuzzer evaluation method reveals the effectiveness of our proposed evaluation method that considers the difficulty of the resolved branches.

Third, we substitute our bandits-based resource allocation mechanism with a simple random scheduling approach (denoted as “random”). Comparing the performance of this variant against BANDFUZZ enables us to quantify the impact of our bandits-based allocation mechanism on the efficacy of the fuzzing process.

We run BANDFUZZ and the three variations on four targets from FuzzBench: `bloaty`, `freetype2`, `woff2ttf`, and `lcms`. These targets were chosen due to their varied sizes and complexities, which provide a comprehensive array of challenges that a fuzzer may face when dealing with different code structures. To elaborate, `bloaty` is a large-scale program with 89,530 edges and is equipped with 94 initial seeds. Conversely, `freetype2` has 19,056 edges but operates with only two seeds, representing a large program with a minimal initial seed count. `woff2ttf` is a smaller program with 10,923 edges and 62 initial seeds. Finally, `lcms` is the smallest program with 6,959 edges and just one seed, representing a compact program with a sparse seed allocation.

**Experiment IV: Effectiveness of the customized bandits model.** Recall that our customized bandits model introduces two key designs, reward discretization and periodic parameter reset. Here, reward discretization cannot be changed because, without a discrete reward, we cannot use the TS algorithm for our method. As such, to address RQ4, we remove the periodic parameter reset mechanism (denoted as “no-reset”) and run in on the four targets used in Experiment III. Comparing this variation with BANDFUZZ can demonstrate the efficacy of considering the dynamical changes in bandits’ reward distribution.

**Experiment V: Hyper-parameter sensitivity.** To answer RQ5, we vary two key hyper-parameters of BANDFUZZ: the time budget for each round  $T_I$  and the reset interval  $I_R$ . Specifically, we adjust  $T_I$  between 90 to 180 seconds in 30-second increments and vary  $I_R$  from 90 to 180 minutes at 30-minute intervals. We compare these variations with BANDFUZZ’s default setup on the four targets used in Experiment III & IV to quantify the sensitivity of BANDFUZZ against these hyper-parameters. We conduct a Nemenyi post-hoc test [67], which is widely used to determine whether there are notable result differences among different experimental setups.

## 5.3. Experiment Result

**Experiment I: Comparison with individual fuzzer.** Table 1 shows the average score of each fuzzer across individual target programs, as well as the median and mean score, after a 24-hour testing period. The table first shows that BANDFUZZ achieves the highest median and average score across all the FuzzBench targets. Regarding individual targets, BANDFUZZ outperforms all other individual fuzzers

| Benchmark                          | BANDFUZZ | AFL++  | honggfuzz | LAF-Intel | Radamsa | MOpt   | AFL    | AFLFast | FairFuzz | DARWIN |
|------------------------------------|----------|--------|-----------|-----------|---------|--------|--------|---------|----------|--------|
| bloaty_fuzz_target                 | 97.97    | 97.41  | 77.56     | 96.38     | 82.45   | 94.95  | 93.65  | 93.78   | 80.52    | †68.02 |
| curl_curl_fuzzer_http              | 99.07    | 98.09  | 85.09     | 85.23     | 89.74   | 92.03  | 90.64  | 89.46   | 83.97    | †62.99 |
| freetype2_ftfuzzer                 | 98.15    | 88.38  | 53.91     | 77.56     | 53.46   | 61.38  | 59.50  | 58.66   | 58.42    | †21.27 |
| harfbuzz_hb-shape-fuzzer           | 98.87    | 98.84  | 81.73     | 97.22     | 84.47   | 96.42  | 95.74  | 95.34   | 88.69    | †68.31 |
| jsoncpp_jsoncpp_fuzzer             | 100.00   | 100.00 | 100.00    | 100.00    | 100.00  | 100.00 | 100.00 | 100.00  | 100.00   | 100.00 |
| lcms_cms_transform_fuzzer          | 97.95    | 93.03  | 30.30     | 64.93     | 33.42   | 29.39  | 29.01  | 27.32   | 64.19    | 30.61  |
| libjpeg-turbo_libjpeg_turbo_fuzzer | 82.74    | 82.65  | 82.29     | 82.65     | 82.61   | 82.26  | 82.18  | 82.19   | 82.19    | †81.64 |
| libpcap_fuzz_both                  | 92.32    | 89.85  | 65.84     | 83.64     | 68.78   | 1.08   | 1.08   | 1.08    | 1.08     | †0.00  |
| libpng_libpng_read_fuzzer          | 95.95    | 95.50  | 94.62     | 95.52     | 92.47   | 93.07  | 93.02  | 92.12   | 93.19    | 93.26  |
| libxml2_xml                        | 99.61    | 99.41  | 94.74     | -         | 92.00   | 96.73  | 96.64  | 96.77   | 89.74    | 96.99  |
| libxslt_xpath                      | 98.31    | 98.11  | 93.23     | 49.67     | 90.48   | 89.58  | 90.80  | 90.37   | 92.29    | 92.63  |
| mbdttls_fuzz_dtlsclient            | 74.73    | 74.25  | 71.52     | 50.41     | 64.34   | 71.03  | 70.70  | 68.52   | 73.73    | †49.30 |
| openh264_decoder_fuzzer            | 99.03    | 98.43  | 96.51     | 99.28     | 79.75   | 99.07  | 99.06  | 99.40   | 98.46    | †0.00  |
| openssl_x509                       | 99.95    | 99.94  | 98.97     | 99.59     | 99.29   | 99.69  | 99.73  | 99.72   | 99.69    | 99.91  |
| openthread_ot-ip6-send-fuzzer      | 77.05    | 89.91  | 70.08     | 71.28     | 69.06   | 73.10  | 72.04  | 70.95   | 69.14    | †0.00  |
| proj4_proj_crs_to_crs_fuzzer       | 96.70    | 90.67  | 50.13     | 67.82     | 52.22   | 9.83   | 10.32  | 9.35    | 10.47    | †0.70  |
| re2_fuzzer                         | 99.84    | 99.43  | 98.39     | 99.76     | 98.54   | 98.89  | 99.34  | 98.63   | 99.13    | 99.01  |
| stb_stbi_read_fuzzer               | 96.18    | 95.59  | 92.34     | 90.28     | 89.01   | 89.56  | 90.35  | 89.81   | 89.85    | 90.62  |
| systemd_fuzz-link-parser           | 98.76    | 98.35  | 88.64     | 97.93     | 89.67   | 90.91  | 90.50  | 90.70   | 84.30    | †51.65 |
| vorbis_decode_fuzzer               | 98.79    | 98.40  | 97.62     | 99.38     | 93.75   | 97.58  | 97.74  | 97.70   | 97.54    | †59.25 |
| woff2_convert_woff2ttf_fuzzer      | 98.56    | 97.36  | 89.19     | 99.26     | 82.26   | 90.10  | 88.86  | 89.69   | 81.56    | †62.95 |
| Median Score                       | 98.31    | 97.41  | 88.64     | 90.28     | 84.47   | 90.91  | 90.64  | 90.37   | 84.30    | 62.99  |
| Average Score                      | 95.26    | 94.46  | 81.56     | 81.32     | 80.37   | 78.89  | 78.61  | 78.17   | 78.01    | 58.53  |

TABLE 1: Fuzzer score summary in FuzzBench report. The fuzzers are organized based on average scores, representing their average relative coverage. Those with higher values are positioned toward the left side of the table. A green background signifies a superior score, whereas a blue background denotes the second-best score. Note that in FuzzBench report, any difference less than 1% is considered as a random effect and thus disregarded. The symbol † indicates that the fuzzer encounters a crash before completing its 24-hour task. The symbol - indicates that the fuzzer does not support the target.

| Benchmarks                   | BANDFUZZ-8     |         | autofz-8      |        | autofz        |        |
|------------------------------|----------------|---------|---------------|--------|---------------|--------|
|                              | mean           | std     | mean          | std    | mean          | std    |
| boringssl-2016-02-12         | 1671.7         | 0.48    | 1622.9        | 12.03  | <b>1678.5</b> | 17.06  |
| c-ares-CVE-2016-5180         | 42.9           | 0.32    | <b>43.0</b>   | 0.00   | 42.9          | 0.32   |
| freetype2-2017               | <b>10946.4</b> | 418.56  | 7887.1        | 326.99 | 8236.8        | 234.77 |
| guetzli-2017-3-30            | <b>1409.9</b>  | 3.75    | 1392.9        | 9.39   | 1399.6        | 8.25   |
| harfbuzz-1.3.2               | <b>4866.0</b>  | 41.63   | 4622.8        | 57.72  | 4722.3        | 53.50  |
| json-2017-02-12              | 649.1          | 1.52    | 649.5         | 5.82   | <b>662.9</b>  | 0.32   |
| libarchive-2017-01-04        | <b>5071.1</b>  | 190.69  | 4453.0        | 308.46 | 4687.7        | 300.56 |
| libjpeg-turbo-07-2017        | <b>2028.9</b>  | 91.89   | 1836.7        | 79.63  | 2026.6        | 42.24  |
| libpng-1.2.56                | 1054.1         | 9.54    | 1076.9        | 11.86  | <b>1090.0</b> | 8.82   |
| libssh-2017-1272             | <b>1068.0</b>  | 59.60   | 1006.9        | 14.50  | 1064.9        | 54.02  |
| libxml2-v2.9.2               | <b>8308.8</b>  | 292.49  | 3955.4        | 73.62  | 4406.6        | 353.30 |
| lcms-2017-03-21              | 1923.8         | 179.52  | <b>1991.8</b> | 39.44  | 1976.1        | 37.25  |
| openssl-1.0.1f               | <b>5536.4</b>  | 551.51  | 5058.8        | 822.06 | 5450.7        | 250.61 |
| openssl-1.0.2d               | <b>2136.1</b>  | 1.66    | 2126.4        | 3.37   | 2133.1        | 2.60   |
| openssl-1.1.0c-bignum        | <b>2036.9</b>  | 0.74    | 2034.8        | 0.42   | 2034.7        | 0.82   |
| openssl-1.1.0c-x509          | 6105.8         | 0.63    | 6144.0        | 3.62   | <b>6145.3</b> | 1.70   |
| openthread-2018-02-27-ip6    | 1412.5         | 318.97  | 1280.4        | 323.10 | <b>1412.8</b> | 320.93 |
| openthread-2018-02-27-radio  | <b>2783.7</b>  | 240.00  | 2595.2        | 290.29 | <b>3260.8</b> | 42.63  |
| pcr2-10.00                   | <b>9909.4</b>  | 125.82  | 8956.5        | 73.64  | 9124.1        | 118.82 |
| proj4-2017-08-14             | 2461.4         | 192.53  | 1015.1        | 595.85 | <b>3568.7</b> | 43.01  |
| re2-2014-12-09               | <b>2485.3</b>  | 7.10    | 2470.6        | 8.85   | 2478.7        | 7.24   |
| sqlite-2016-11-14            | <b>1981.0</b>  | 1112.83 | 1551.5        | 147.60 | 1624.0        | 0.00   |
| vorbis-2017-12-11            | <b>1264.1</b>  | 6.94    | 1236.3        | 10.75  | 1242.4        | 12.66  |
| woff2-2016-05-06             | 1071.9         | 23.32   | <b>1097.1</b> | 16.93  | 1062.7        | 33.05  |
| wpantund-2018-02-27          | <b>3315.2</b>  | 99.03   | 3079.3        | 84.01  | 3176.1        | 52.75  |
| Average Coverage Enhancement | -              | -       | 16.0%         | -      | 4.8%          | -      |
| Average Score                | 91.5           | -       | 84.3          | -      | 89.6          | -      |

TABLE 2: The branch coverage of BANDFUZZ vs. autofz-8 and full autofz. Each row displays the arithmetic mean and standard deviation of branch coverage for every fuzzer. The maximum branch coverage achieved for each target is also highlighted.

in 18 of the 21 evaluated targets. This result validates BANDFUZZ’s capability in harnessing and ensembling the unique strengths of individual fuzzers. In other words, BANDFUZZ’s bandit algorithm is able to consistently allocate more resources to fuzzers with the best performance along the fuzzing process. As a result, it puts together an ensemble

strategy with the best performance.

Figure 8 in the Appendix shows the branch coverage of BANDFUZZ vs individual fuzzers across the time, we can observe that BANDFUZZ is more stable than individual fuzzers. This is another advantage of being an ensemble strategy in that BANDFUZZ can always choose a functional fuzzer and will not be largely affected by the failure of one or two fuzzers. The fuzzer management feature introduced in §4.2 also helps the stability. This feature enables BANDFUZZ to promptly identify and address any issues with the fuzzers during operation, ensuring consistent performance.

However, it is noteworthy that despite its overall superior performance, BANDFUZZ occasionally exhibits a slower initial growth in code coverage during the initial 30 to 60 minutes compared to certain individual fuzzers (Figure 8 ). We believe this is an expected characteristic of BANDFUZZ’s operation, as BANDFUZZ necessitates multiple iterations to effectively explore and determine the dynamic effectiveness of the various fuzzers it includes.

Finally, we notice that the coverage of score of BANDFUZZ demonstrates relatively marginal improvement than AFL++. This is mainly because AFL++ (with *cmplog* enabled) performs much better than other fuzzers in most targets. By analyzing the seeds, we found that AFL++ dominates other fuzzers in code coverage and is also assigned the most resources. This demonstrates that BANDFUZZ indeed identifies the best fuzzer and allocates most resources to it. Furthermore, we believe the similar performance of BANDFUZZ and AF++ does not dilute the necessity of BANDFUZZ for the following reasons. First, as demonstrated in §6, BANDFUZZ performs much better than the state-of-the-art fuzzers in mutation scores, another widely used metric for

evaluating a fuzzer’s ability to find vulnerabilities. Second, as an ensemble approach, BANDFUZZ is more stable than individual fuzzers like AFL++. Third, in scenarios where AFL++ under-performs, BANDFUZZ can still choose other superior fuzzers and achieve decent and stable performance.

**Experiment II: Comparison with autofz.** Table 2 presents a comparison between BANDFUZZ-8, autofz-8, and autofz on FTS targets. First, the table shows that BANDFUZZ-8 outperforms autofz-8 in terms of average branch coverage, achieving an improvement of 16.0%. More specifically, BANDFUZZ outperforms autofz-8 in 19 of the 25 evaluated programs with a notable margin in branch coverage. This superior performance validates the effectiveness of our advanced collaboration strategy, including our real-time seed synchronization, faithful and efficient fuzzer evaluation, and bandits-based resource allocation/fuzzer scheduling strategy. In particular, it verifies our discussion in §2 and §3 that autofz only realizes greedy resource allocation strategies while BANDFUZZ can learn globally more optimal strategies.

It is also worth noting that in cases where BANDFUZZ-8 does not outperform autofz-8, the decline in coverage is relatively marginal (not exceeding 3.5%). The standard deviation of branch coverage for BANDFUZZ-8 in these specific targets aligns closely with that of autofz-8. Furthermore, it is important to highlight an outlier occurrence, where BANDFUZZ-8’s standard deviation on the target `sqlite-2016-11-14` was exceptionally high, because of an isolated trial that yielded remarkably high branch coverage. This result shows that BANDFUZZ has a higher stability than autofz.

Moreover, BANDFUZZ also slightly outperforms the original autofz, which incorporates three additional fuzzers (including some hybrid fuzzers). Despite having fewer fuzzers, BANDFUZZ-8 achieves an average branch coverage that exceeds that of the original autofz by 4.8%. Although BANDFUZZ does not integrate hybrid fuzzers (with concolic execution), powered with our advanced resource allocation method, BANDFUZZ is superior to autofz, which integrates these advanced fuzzers. This further demonstrates the importance of our bandits-based resource allocation method.

**Experiment III: Ablation study.** Figure 4 shows the ablation study results. First, BANDFUZZ outperforms “no-sync” by an average of 9.6% in branch coverage. This improvement validates the effectiveness of our real-time seed synchronization mechanism. It prevents individual fuzzers from repeatedly covering the same branches, enabling a more thorough and efficient exploration of the entire program space.

However, an exception is observed where BANDFUZZ does not demonstrate superiority over “no-sync” in the case of `woff2`. This deviation is because of the unique characteristics of the target - a sparse edge count combined with a large number of initial seeds. In such cases, the advantage derived from inter-fuzzer knowledge exchange becomes less significant since each fuzzer can already have a comprehensive exploration of the program’s structure.

Second, Figure 4 also shows that BANDFUZZ outperforms “naive-reward” by about 5.8% in branch coverage.

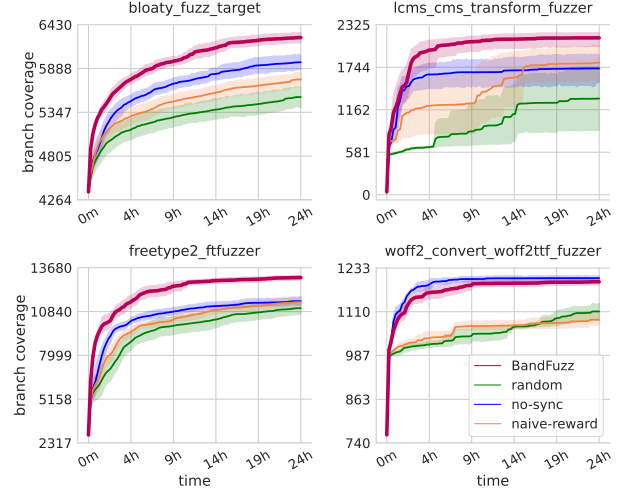


Figure 4: Comparison among BANDFUZZ and two variants. Each line plot represents the arithmetic mean along with a 95% confidence interval, derived from 10 independent trials. The label *random* refers to BANDFUZZ that employs random resource allocation in each round, and *no-sync* represents BANDFUZZ operating without seed synchronization.

This result verifies the importance of our proposed fuzzer evaluation mechanism. As discussed in §3.1, assigning rewards solely based on coverage is limited in that it cannot account for the difficulty and complexity of each new coverage, resulting in a less comprehensive assessment of fuzzer performance.

Last but not least, the “random” variation triggers the largest performance drop, recording about a 22.0% reduction in branch coverage. This highlights the importance of our bandits-powered resource allocation strategy, which explore global optimal solutions for resource allocation, substantially optimizing fuzzing efficiency.

**Experiment IV: Effectiveness of the customized bandits model.** Figure 5 demonstrates the comparison between BANDFUZZ and the “no-reset” version of BANDFUZZ, which excludes the parameter reset operation in our customized bandits model. The figure shows an 8.1% decrease in branch coverage for “no-reset”. This result verifies the necessity of our customized bandits model. As discussed in §3.3, without a reset mechanism, BANDFUZZ’ cannot mitigate the impact of outdated information. As a result, “no-reset” cannot precisely adapt to changes in reward distributions and thus has limited fuzzing effectiveness.

**Experiment V: Hyper-parameter sensitivity.** Figure 6 depicts the performance differences introduced by varying two key hyper-parameters of BANDFUZZ:  $T_I$  and  $I_R$ . The figure shows that changing these two hyper-parameters introduces only marginal changes in overall fuzzing effectiveness. The result verifies the insensitivity of BANDFUZZ against key hyper-parameter changes. This also demonstrates the practicality of BANDFUZZ, as users are not required to carefully choose the optimal hyper-parameters.

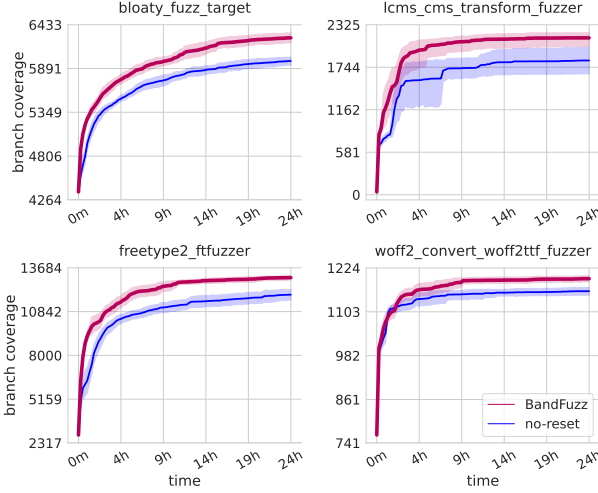


Figure 5: Comparison among BANDFUZZ and two variants. Each line plot represents the arithmetic mean along with a 95% confidence interval, derived from 10 independent trials. The label *no-reset* refers to BANDFUZZ without reset mechanism, and *naive-reward* represents BANDFUZZ adopting a naive algorithm to evaluate fuzzers.

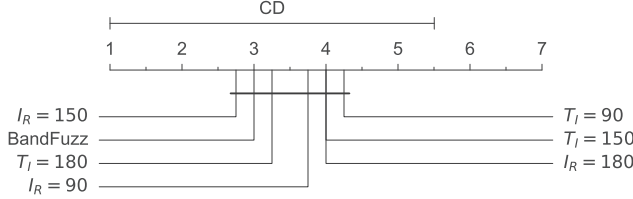


Figure 6: Critical Difference (CD) among various configurations of BANDFUZZ. The average ranking of fuzzers is represented by each number. The horizontal line indicates no critical difference in performance among the grouped fuzzers, as determined by the Nemenyi post-hoc test. Each label represents a variation of the depicted hyperparameter while keeping another parameter at its default values.

## 6. Real-World Assessment

Our evaluation of BANDFUZZ extends beyond controlled experiments to its application in a recent, highly competitive fuzz testing event, SBFT 2024, as detailed in [68]. This event presents a modified FuzzBench platform for evaluating advanced fuzzing techniques in a real-world context.

During the competition, participant fuzzers, including BANDFUZZ, are deployed on the modified FuzzBench platform with the same amount of resources for the same period of time. Notably, the evaluation metric is different from our experiments in §5, which primarily utilize branch coverage as the performance metric. Instead, the competition organizers employ the *mutation analysis* as the primary evaluation tool, a method that is increasingly used as an alternative to traditional coverage metrics in fuzzing benchmarks, as discussed in [69].

Mutation analysis introduces synthetic faults into target programs to create program mutants, minimizing human

TABLE 3: Number of mutants killed per benchmark, with each column reflecting the total mutants killed by the respective fuzzer.

| Targets                     | BANDFUZZ    | FishFuzz | AF1         | Pastis      | AF2  | libAFL | libFuzzer |
|-----------------------------|-------------|----------|-------------|-------------|------|--------|-----------|
| freetype2_ftfuzzer          | <b>6814</b> | 5924     | 5944        | 6158        | 5900 | 6172   | 5131      |
| jsoncpp_jsoncpp_fuzzer      | <b>1150</b> | 672      | 650         | 686         | 651  | 638    | 679       |
| lcms cms_transform_fuzzer   | <b>1709</b> | 1461     | 1522        | 1560        | 1570 | 1547   | 1527      |
| libpcap_fuzz_both           | 2095        | 2096     | <b>2443</b> | 1852        | 2031 | 1811   | 1636      |
| libxml2_xml                 | <b>7259</b> | 6582     | 6880        | 6240        | 6212 | 6160   | 6125      |
| re2_fuzzer                  | <b>6651</b> | 5967     | 3672        | 3670        | 3650 | 3635   | 3687      |
| stb_stbi_read_fuzzer        | 1597        | 1643     | 1550        | <b>1647</b> | 1592 | 1562   | 1474      |
| zlib_zlib_uncompress_fuzzer | <b>361</b>  | 333      | 352         | 359         | 326  | 326    | 355       |
| Average Mutation Score      | <b>98</b>   | 86       | 84          | 82          | 81   | 79     | 77        |

bias in the process. A mutant is deemed “killed” if the fuzzer generates at least one input causing a deviation from the original program’s behavior (i.e., trigger a synthetic fault). The effectiveness of a fuzzer is thus quantified by the number of mutants it successfully killed, with a higher count indicative of superior performance. As discussed in [69], this metric offers a more robust and comprehensive assessment compared to coverage by addressing issues like saturation and over-fitting. Moreover, by integrating a variety of fault types into the program, mutation analysis provides a broader spectrum evaluation of a fuzzer’s capability to discover vulnerabilities, which is the main purpose of fuzzing in the security context.

The mutation analysis results, as presented in Table 3, clearly demonstrate BANDFUZZ’s outstanding performance in the competition. It surpassed not only widely recognized fuzzers like libAFL [70] and libFuzzer [71] but also advanced contenders such as FishFuzz [66] and Pastis [72], alongside two anonymously fuzzers denoted as *AF1* and *AF2*. Specifically, BANDFUZZ achieved the highest count of killed mutants in six out of the eight targeted categories. Moreover, while the other fuzzers showed relatively narrow variations in their average mutation scores (ranging from 79 to 86), BANDFUZZ recorded the highest average, a significant 98, marking a substantial lead over its competitors. The mutation score is calculated by taking the median number of mutants it killed and dividing it by the highest number of mutants killed recorded for the same target. In comparison to the second-ranked fuzzer, TuneFuzz [66], BANDFUZZ’s average mutation score is higher by an impressive 14%. Overall, this evaluation demonstrates the significant superiority of BANDFUZZ over other SOTA fuzzers (widely used in academia or industry) in *bug detection capability for complex, real-world programs*.

## 7. Related Work

In addition to heuristic-based and program analysis-based fuzzers mentioned in §2.1, recent research also explores learning-based fuzzers [13], [6], [64], [73], [74], [75], [76], [77], [78], [79], [1], [2], [80], [81], [82], [83], [11], [84], [85]. These methods leverage a wide range of ML methods, such as deep learning and reinforcement learning, to enhance specific components of existing fuzzers or to construct new fuzzing frameworks. Among these methods, a notable line of research utilizes multi-arm bandits to improve existing fuzzers [13], [6], [86], [64], [65], [87], [88]. In what follows, we summarize these methods and discuss their limitations.

**Improving seed selection.** Existing bandits-based seed selection methods [13], [6], [86], [64] typically treat the seeds as the arms and employ standard bandit algorithms, such as Upper Confidence Bound (UCB) [59], to dynamically adjust the weight of each seed based on the fuzzing feedback. While these techniques outperform heuristic-based strategies, they face two key limitations. First, dealing with a large volume of seeds results in bandits having a large number of arms, leading to a less efficient learning process. Furthermore, the continual growth in the number of seeds violates the fundamental assumption of bandits, which is designed for a fixed number of arms. Applying bandit algorithms without addressing these issues may lead to performance improvements that are not universally applicable across different test scenarios.

**Enhancing power scheduling.** Beyond seed selection, recent studies have also applied bandit algorithms to enhance power scheduling. For example, Zhang et al. [65] treat seeds as the arms of bandits. However, a notable difference is that researchers employ the UCB algorithm to dynamically optimize not just the selection of seeds but also the distribution of energy. Consequently, bandits-based power scheduling encounters similar challenges as those faced by bandits-based seed selection methods.

**Optimizing input mutation.** Recent research also leverages bandits for mutation scheduling [73], [87], [21], [89], [88]. For example, Wu et al. [87] designed a two-layer bandits. Here, the arms of the first layer are the number of mutators to be used and the second layer’s arms are types of mutators. This method leverages the UCB1-Tuned algorithm [59] to select arms for both layers to guide mutator selection. This method, although not a typical bandits design, demonstrates a decent empirical performance.

In this work, rather than use it for improving individual components, we leverage multi-arm bandits to build a collaborative fuzzing framework, which is more robust and generic than individual fuzzers.

## 8. Discussion

**More fuzzer support.** BANDFUZZ incorporates 10 state-of-the-art fuzzers, which employ advanced strategies to enhance nearly every fuzzing component. Hybrid Fuzzing is the only main category of fuzzers that BANDFUZZ has not employed. These fuzzers incorporate concolic (**concrete symbolic**) executors to improve the fuzzers’ ability to solve difficult constraints. Integrating concolic execution into BANDFUZZ presents several challenges.

First, synchronizing seeds between concolic executors and other fuzzers cannot be simply achieved via copying the global seed pool, as concolic executors are not aware of the coverage of fuzzers. Without synchronization, concolic executors will explore program states already founded by other fuzzers, leading to significant resource wastage. Additionally, given that concolic executors are much slower than regular fuzzers, it is difficult to maintain a fair resource allocation between fuzzers and concolic executors while capturing fuzzer dynamics in real-time.

Tackling these challenges requires a comprehensive update of BANDFUZZ, including designing a new seed synchronization and fuzzer evaluation mechanism, which demands non-trivial efforts. Our primary goal is to investigate an effective collaborative fuzzing strategy. Additionally, we position BANDFUZZ as the preliminary effort toward comprehensive collaborative fuzzing frameworks. As such, we defer to integrating hybrid fuzzers as our future work.

**Alternative seed assessment.** Recall that BANDFUZZ employs code coverage and the interval between branch discoveries as metrics to assess the quality of new seeds. However, there exist additional metrics that could be used to evaluate seed quality, such as basic-block centrality [90] and the distance to sanitizer [66]. In future work, we plan to explore other seed quality assessment metrics and refine our fuzzer evaluation accordingly.

**TS improvement.** Recall that we developed a customized Thompson Sampling (TS) approach for our bandits model, incorporating reward discretization to manage continuous rewards and a reset mechanism to tackle non-stationary reward distributions. Although these modifications are effective, there is room for improvement or replacement with alternative techniques. For instance, we could evaluate different discretization strategies or reformulate TS by treating rewards as a continuous distribution rather than a discrete Bernoulli distribution. Furthermore, exploring adversarial bandits, such as EXP3 [91], which are specifically designed to handle non-stationary reward distributions, could offer additional enhancements to our model.

**Efficient compilation.** As a collaborative fuzzing framework, BANDFUZZ is designed to integrate with any type of fuzzer. Technically, the fuzzers incorporated in BANDFUZZ might employ various instrumentation techniques, necessitating multiple compilations of the target program, each with a distinct instrumentation pass. This process results in greater compilation overhead compared to using individual fuzzers. In our future research efforts, we plan to investigate more efficient instrumentation strategies to mitigate the overhead associated with diverse instrumentation requirements.

**More sophisticated targets.** The current version of BANDFUZZ is optimized for the C/C++ programs in the Linux user space. Our future work will extend BANDFUZZ to other platforms and fuzzing domains, such as kernel fuzzing [92], [77], [51] and compiler fuzzing [93], [85], [79], [94]. This extension will potentially demonstrate the effectiveness of BANDFUZZ’s collaboration strategy across various software layers and development tools, broadening the impact of collaborative fuzzing in the software security landscape.

## 9. Conclusion

In this paper, we introduce BANDFUZZ, a novel collaborative fuzzing framework powered by a customized bandits model. Different from the greedy method utilized by the state-of-the-art collaborative fuzzing framework, *autofz*, BANDFUZZ model the long-term impact of individual fuzzers, learning globally optimal collaborative strategies. Together with the bandits-based resource allocation, we

also propose a new metric for fuzzer evaluation, a real-time seed synchronization mechanism, as well as a set of implementation-wise optimizations. Through extensive experiments, we first demonstrate BANDFUZZ’s superior effectiveness, stability, and adaptability across a variety of programs, compared to autofz and widely used individual fuzzers. Moreover, we conduct a comprehensive ablation study and hyper-parameter sensitivity test to verify BANDFUZZ’s key designs and its insensitivity to the changes in hyper-parameters. Finally, we demonstrate BANDFUZZ’s effectiveness in bug detection for real-world programs by analyzing the results of a recent fuzzing competition, where BANDFUZZ won the first place. Through these findings, we safely conclude that collaborative fuzzing presents a promising solution for constructing generic fuzzers, and multi-arm bandits can be used to design effective collaborative fuzzing strategies. We hope our work can inspire further advancements in collaborative fuzzing through multi-arm bandits and reinforcement learning, to push the boundaries of software fuzzing in real-world applications.

## References

- [1] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, “Neuzz: Efficient fuzzing with neural program smoothing,” in *Proc. of S&P*, 2019.
- [2] D. She, R. Krishna, L. Yan, S. Jana, and B. Ray, “Mtfuzz: fuzzing with a multi-task neural network,” in *Proc. of ESEC/FSE*, 2020.
- [3] Y. Li, Y. Xue, H. Chen, X. Wu, C. Zhang, X. Xie, H. Wang, and Y. Liu, “Cerebro: context-aware adaptive fuzzing for effective vulnerability detection,” in *Proc. of ESEC/FSE*, 2019.
- [4] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su, “Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization,” in *Proc. of NDSS*, 2020.
- [5] Y. Zhao, X. Wang, L. Zhao, Y. Cheng, and H. Yin, “Alphuzz: Monte carlo search on seed-mutation tree for coverage-guided fuzzing,” in *Proc. of ACSAC*, 2022.
- [6] J. Wang, C. Song, and H. Yin, “Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing,” *Journal of Systems and Software*, 2021.
- [7] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, “Optimizing seed selection for fuzzing,” in *Proc. of USENIX Security*, 2014.
- [8] J. Liang, Y. Jiang, M. Wang, X. Jiao, Y. Chen, H. Song, and K.-K. R. Choo, “Deepfuzzer: Accelerated deep greybox fuzzing,” *IEEE Transactions on Dependable and Secure Computing*, 2019.
- [9] A. Herrera, H. Gunadi, S. Magrath, M. Norrish, M. Payer, and A. L. Hosking, “Seed selection for successful fuzzing,” in *Proc. of ISSTA*, 2021.
- [10] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, “Collafl: Path sensitive fuzzing,” in *Proc. of S&P*, 2018.
- [11] S. Zhu, J. Wang, J. Sun, J. Yang, X. Lin, T. Wang, L. Zhang, and P. Cheng, “Better pay attention whilst fuzzing,” *IEEE Transactions on Software Engineering*, 2023.
- [12] M. Böhme, V.-T. Pham, and A. Roychoudhury, “Coverage-based greybox fuzzing as markov chain,” in *Proc. of CCS*, 2016.
- [13] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou, “Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit,” in *Proc. of USENIX Security*, 2020.
- [14] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, “Vuzzer: Application-aware evolutionary fuzzing,” in *Proc. of NDSS*, 2017.
- [15] C. Lemieux and K. Sen, “Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage,” in *Proc. of ASE*, 2018.
- [16] P. Jauernig, D. Jakobovic, S. Picek, E. Stapf, and A.-R. Sadeghi, “Darwin: Survival of the fittest fuzzing mutators,” in *Proc. of NDSS*, 2023.
- [17] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, and Y. Song, “Mopt: Optimized mutation scheduling for fuzzers,” in *Proc. of USENIX Security*, 2022.
- [18] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, “Redqueen: Fuzzing with input-to-state correspondence,” in *Proc. of NDSS*, 2019.
- [19] “Aki helin / radamsa · gitlab,” <https://gitlab.com/akihe/radamsa>, 2023.
- [20] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang, “Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery,” in *Proc. of S&P*, 2019.
- [21] X. Wang, C. Hu, R. Ma, D. Tian, and J. He, “Cmfuzz: context-aware adaptive mutation for fuzzers,” *Empirical Software Engineering*, 2021.
- [22] M. Lee, S. Cha, and H. Oh, “Learning seed-adaptive mutation strategies for greybox fuzzing,” in *Proc. of ICSE*, 2023.

- [23] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz, "Sok: Prudent evaluation practices for fuzzing," in *Proc. of S&P*, 2024.
- [24] "Fuzzbench: 2023-12-15-aflpp report," <https://fuzzbench.com/reports/experimental/2023-12-15-aflpp/index.html>, 2023.
- [25] D. Liu, J. Metzman, M. Böhme, O. Chang, and A. Arya, "Sbft tool competition 2023-fuzzing track," *ArXiv Preprint*, 2023.
- [26] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++: Combining incremental steps of fuzzing research," in *Proc. of WOOT*, 2020.
- [27] "Fuzzbench reports," <https://fuzzbench.com/reports/experimental/>, 2024.
- [28] "Sbft 2023 fuzzing competition," <https://sbft23.github.io/tools/fuzzing>, 2023.
- [29] S. Österlund, E. Geretto, A. Jemmett, E. Güler, P. Göz, T. Holz, C. Giuffrida, and H. Bos, "Collabfuzz: A framework for collaborative fuzzing," in *Proc. of EuroSec*, 2021.
- [30] E. Güler, P. Göz, E. Geretto, A. Jemmett, S. Österlund, H. Bos, C. Giuffrida, and T. Holz, "Cupid: Automatic fuzzer selection for collaborative fuzzing," in *Proc. of ACSAC*, 2020.
- [31] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *Proc. of USENIX Security*, 2019.
- [32] Y.-F. Fu, J. Lee, and T. Kim, "autofz: Automated fuzzer composition at runtime," in *Proc. of USENIX Security*, 2023.
- [33] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "Fuzzbench: an open fuzzer benchmarking platform and service," in *Proc. of ESEC/FSE*, 2021.
- [34] Google, "Fuzzer test suite (fts)," <https://opensource.google/projects/fuzzer-test-suite>, 2021.
- [35] M. Zalewski, "American fuzzy lop," <http://lcamtuf.coredump.cx/afl/>, 2013.
- [36] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proc. of S&P*, 2018.
- [37] L. Zhao, Y. Duan, H. Yin, and J. Xuan, "Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing," in *Proc. of NDSS*, 2019.
- [38] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, 2019.
- [39] K. Zhang, X. Xiao, X. Zhu, R. Sun, M. Xue, and S. Wen, "Path transitions tell more: Optimizing fuzzing schedules via runtime program states," in *Proc. of ICSE*, 2022.
- [40] J. Chen, S. Wang, S. Cai, C. Zhang, H. Chen, J. Chen, and J. Zhang, "A novel coverage-guided greybox fuzzing based on power schedule optimization with time complexity," in *Proc. of ASE*, 2022.
- [41] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proc. of NDSS*, 2016.
- [42] J. Chen, W. Han, M. Yin, H. Zeng, C. Song, B. Lee, H. Yin, and I. Shin, "Symsan: Time and space efficient concolic execution via dynamic data-flow analysis," in *Proc. of USENIX Security*, 2022.
- [43] S. Poeplau and A. Francillon, "Symbolic execution with symcc: Don't interpret, compile!" in *Proc. of USENIX Security*, 2020.
- [44] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "Qsym: A practical concolic execution engine tailored for hybrid fuzzing," in *Proc. of USENIX Security*, 2018.
- [45] S. Poeplau and A. Francillon, "Symqemu: Compilation-based symbolic execution for binaries," in *Proc. of NDSS*, 2021.
- [46] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang, "Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction," in *Proc. of S&P*, 2020.
- [47] L. Borzacchiello, E. Coppa, and C. Demetrescu, "Fuzzolic: mixing fuzzing and concolic execution," *Computers & Security*, 2021.
- [48] Z. Shen, R. Roongta, and B. Dolan-Gavitt, "Drifuzz: Harvesting bugs in device drivers from golden seeds," in *Proc. of USENIX Security*, 2022.
- [49] J. Liang, M. Wang, C. Zhou, Z. Wu, Y. Jiang, J. Liu, Z. Liu, and J. Sun, "Pata: Fuzzing with path aware taint analysis," in *Proc. of S&P*, 2022.
- [50] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "Greyone: Data flow sensitive fuzzing," in *Proc. of USENIX Security*, 2020.
- [51] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "Hfl: Hybrid fuzzing on the linux kernel," in *Proc. of NDSS*, 2020.
- [52] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proc. of CCS*, 2017.
- [53] C. Lemieux, R. Padhye, K. Sen, and D. Song, "Perffuzz: Automatically generating pathological inputs," in *Proc. of ISSTA*, 2018.
- [54] Y. Kim and J. Yoon, "Maxafl: Maximizing code coverage with a gradient-based optimization technique," *Electronics*, 2020.
- [55] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE Transactions on Evolutionary Computation*, 1997.
- [56] D. J. Russo, B. Van Roy, A. Kazerouni, I. Osband, Z. Wen *et al.*, "A tutorial on thompson sampling," *Foundations and Trends in Machine Learning*, 2018.
- [57] W. R. Thompson, "On the likelihood that one unknown probability exceeds another in view of the evidence of two samples," *Biometrika*, 1933.
- [58] Y. Zhang, S. Wang, and Z. Fang, "Matching in multi-arm bandit with collision," in *Proc. of NeurIPS*, 2022.
- [59] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine Learning*, 2002.
- [60] O. Chapelle and L. Li, "An empirical evaluation of thompson sampling," in *Proc. of NeurIPS*, 2011.
- [61] S. Agrawal and N. Goyal, "Analysis of thompson sampling for the multi-armed bandit problem," in *Proc. of COLT*, 2012.
- [62] Google, "Honggfuzz: Security oriented software fuzzer," <https://github.com/google/honggfuzz>, 2020.
- [63] "Laf-intel," <https://lafintel.wordpress.com/>, 2016.
- [64] C. Lyu, H. Liang, S. Ji, X. Zhang, B. Zhao, M. Han, Y. Li, Z. Wang, W. Wang, and R. Beyah, "Slime: program-sensitive energy allocation for fuzzing," in *Proc. of ISSTA*, 2022.
- [65] G. Zhang, P. Wang, T. Yue, X. Kong, S. Huang, X. Zhou, and K. Lu, "Mobfuzz: Adaptive multi-objective optimization in gray-box fuzzing," in *Proc. of NDSS*, 2022.
- [66] H. Zheng, J. Zhang, Y. Huang, Z. Ren, H. Wang, C. Cao, Y. Zhang, F. Toffalini, and M. Payer, "Fishfuzz: Catch deeper bugs by throwing larger nets," in *Proc. of USENIX Security*, 2023.
- [67] M. Hollander, D. A. Wolfe, and E. Chicken, *Nonparametric statistical methods*. John Wiley & Sons, 2013.
- [68] "Sbft 2024 fuzzing competition," <https://sbft24.github.io/tools/fuzzing>, 2024.
- [69] P. Göz, B. Mathis, K. Hassler, E. Güler, T. Holz, A. Zeller, and R. Gopinath, "Systematic assessment of fuzzers using mutation analysis," in *Proc. of USENIX Security*, 2023.
- [70] A. Fioraldi, D. C. Maier, D. Zhang, and D. Balzarotti, "Libafl: A framework to build modular and reusable fuzzers," in *Proc. of CCS*, 2022.

- [71] “libfuzzer – a library for coverage-guided fuzz testing,” <https://lsvm.org/docs/LibFuzzer.html>, 2018.
- [72] R. David, R. Abou Chaaya, and C. Heitman, “Pastis: A collaborative approach to combine heterogeneous software testing techniques,” in *Proc. of SBFT*, 2023.
- [73] Y. Koike, H. Katsura, H. Yakura, and Y. Kurogome, “Slopt: Bandit optimization framework for mutation-based fuzzing,” in *Proc. of ACSAC*, 2022.
- [74] J. Yu, X. Lin, and X. Xing, “Gptfuzzer: Red teaming large language models with auto-generated jailbreak prompts,” *ArXiv Preprint*, 2023.
- [75] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, “Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models,” in *Proc. of ISSTA*, 2023.
- [76] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, “Fuzz4all: Universal fuzzing with large language models,” *ArXiv Preprint*, 2024.
- [77] D. Wang, Z. Zhang, H. Zhang, Z. Qian, S. V. Krishnamurthy, and N. Abu-Ghazaleh, “Syzvegas: Beating kernel fuzzing odds with reinforcement learning,” in *Proc. of USENIX Security*, 2021.
- [78] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen, “Fuzz-guard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning,” in *Proc. of USENIX Security*, 2020.
- [79] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, “Compiler fuzzing through deep learning,” in *Proc. of ISSTA*, 2018.
- [80] M. Wu, L. Jiang, J. Xiang, Y. Zhang, G. Yang, H. Ma, S. Nie, S. Wu, H. Cui, and L. Zhang, “Evaluating and improving neural program-smoothing-based fuzzing,” in *Proc. of ICSE*, 2022.
- [81] M.-I. Nicolae, M. Eisele, and A. Zeller, “Revisiting neural program smoothing for fuzzing,” in *Proc. of ESEC/FSE*, 2023.
- [82] K. Böttinger, P. Godefroid, and R. Singh, “Deep reinforcement fuzzing,” in *Proc. of S&PW*, 2018.
- [83] X. Liu, X. Li, R. Prajapati, and D. Wu, “Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing,” in *Proc. of AAAI*, 2019.
- [84] W. Chen, H. Wang, W. Gu, and S. Wang, “Rltrace: Synthesizing high-quality system call traces for os fuzz testing,” in *Proc. of InfoSec*, 2023.
- [85] X. Li, X. Liu, L. Chen, R. Prajapati, and D. Wu, “Alphaprog: reinforcement generation of valid programs for compiler fuzzing,” in *Proc. of AAAI*, 2022.
- [86] Y. Zhao, X. Wang, L. Zhao, Y. Cheng, and H. Yin, “Evolutionary mutation-based fuzzing as monte carlo tree search,” *ArXiv Preprint*, 2021.
- [87] M. Wu, L. Jiang, J. Xiang, Y. Huang, H. Cui, L. Zhang, and Y. Zhang, “One fuzzing strategy to rule them all,” in *Proc. of ICSE*, 2022.
- [88] J. Scott, T. Sudula, H. Rehman, F. Mora, and V. Ganesh, “Banditfuzz: fuzzing smt solvers with multi-agent reinforcement learning,” in *Proc. of ISFM*, 2021.
- [89] S. Karamcheti, G. Mann, and D. Rosenberg, “Adaptive grey-box fuzz-testing with thompson sampling,” in *Proc. of AI & Security*, 2018.
- [90] D. She, A. Shah, and S. Jana, “Effective seed scheduling for fuzzing with graph centrality analysis,” in *Proc. of S&P*, 2022.
- [91] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire, “The non-stochastic multiarmed bandit problem,” *SIAM Journal on Computing*, 2002.
- [92] Google, “syzkaller,” <https://github.com/google/syzkaller>, 2016.
- [93] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, “A survey of compiler testing,” *Acm Computing Surveys (Csur)*, 2020.
- [94] H. Xu, Y. Wang, S. Fan, P. Xie, and A. Liu, “Dsmith: Compiler fuzzing through generative deep learning model with attention,” in *Proc. of IJCNN*, 2020.

## Appendix

### 1. Example of customized multi-arm bandit algorithm and resource allocation

We present a comprehensive explanation of our customized multi-arm bandits, accompanied by a straightforward example that demonstrates the shift in weight distribution.

Let’s assume that we use the method in §3.1 to schedule three fuzzers throughout the fuzzing campaign: AFL, AFLFast and AFL++. At the beginning of a new round, say round  $t$ , the weight distribution for each fuzzer are as follows:  $AFL \sim Beta(2, 2)$ ,  $AFLFast \sim Beta(3, 4)$ ,  $AFL++ \sim Beta(5, 4)$ . For each fuzzer, we sample a value  $\theta$  from its weight distribution. Assume we got  $\theta_{AFL} = 0.57$ ,  $\theta_{AFLFast} = 0.32$ ,  $\theta_{AFL++} = 0.81$ . Since AFL++ has the highest sampled value, it is selected for the round to execute the fuzzing task. After it finishes the task, we need to evaluate its performance by collecting the seeds it generated in this round. Then, we use Algorithm 1 to evaluate those seeds and get a reward value  $r$ , let’s say  $r = 0.78$ . We then need to discretize this value by sampling from a bernoulli distribution  $Bernoulli(p = 0.78)$ , and use the sampled value as the discretized reward. Assume we got 1, then AFL++ obtains a reward equal to 1. The reward distribution of AFL++ is then updated from  $Beta(5, 4)$  to  $Beta(6, 4)$ .

The procedure offers the advantage of achieving a balance between exploration and exploitation. In real world scenarios, scheduling among fuzzers can be highly dynamic. For instance, Figure 7 presents a graphical representation of which fuzzer attains the maximum mathematical expected sampling value in each round. As depicted in the figure, no single fuzzer consistently outperforms others. AFL++, MOpt, and AFLFast each take turns leading in performance.

### 2. More FuzzBench Results

In Figure 8, we show the branch coverage of BAND-FUZZ vs individual fuzzers across the time. The results are consistent with Table 1.

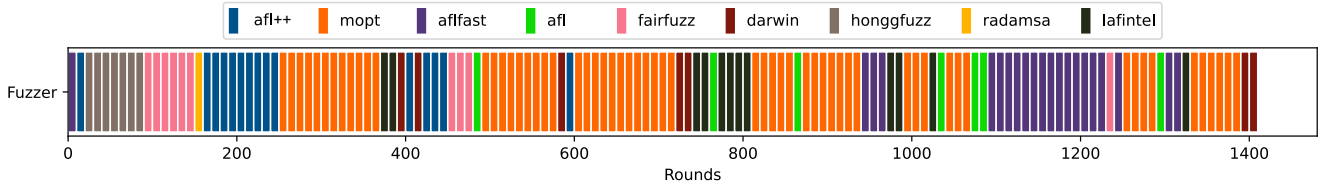


Figure 7: Bar graph representation of fuzzers achieving the highest mathematical expectation for reward distribution over consecutive rounds, sampled every 10 rounds, in a single trial targeting `lcms_cms_transform_fuzzer`. Each colored bar indicates a fuzzer with the leading expectation value at that specific round.

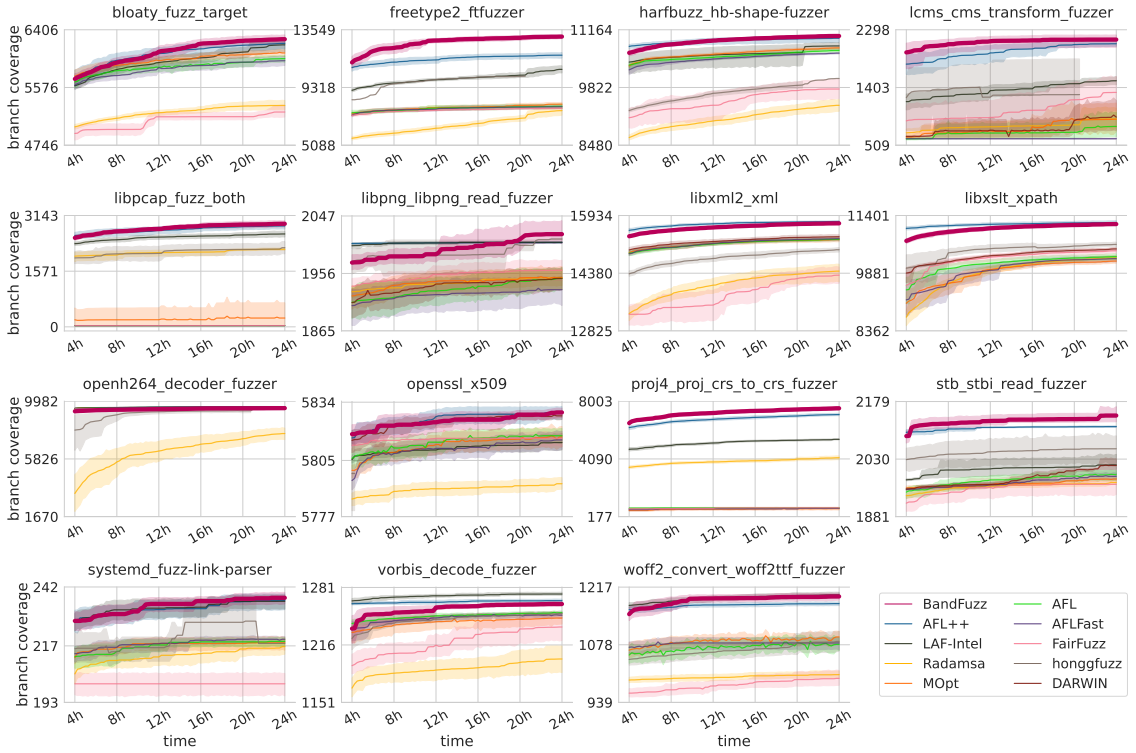


Figure 8: Evaluation of BANDFUZZ and individual fuzzers on FuzzBench: Each line plot in the graph represents the arithmetic mean along with a 95% confidence interval, derived from 10 independent trials. “Branch coverage” denotes the count of branches that each fuzzer has explored.