xFUZZ: A Flexible Framework for Fine-Grained, Runtime-Adaptive Fuzzing Strategy Composition

DONGSONG YU, Zhongguancun Laboratory, China
YIYI WANG[†], Tsinghua University, China and Huazhong University of Science and Technology, China
CHAO ZHANG[*†], Tsinghua University, China and Zhongguancun Laboratory, China
YANG LAN, Zhongguancun Laboratory, China
ZHIYUAN JIANG, National University of Defense Technology, China
SHUITAO GAN, Laboratory for Advanced Computing and Intelligence Engineering, China
ZHEYU MA[†], Tsinghua University, China
WENDE TAN[†], Tsinghua University, China

Fuzzing is one of the most efficient techniques for detecting vulnerabilities in software. Existing approaches struggle with performance inconsistencies across different targets and rely on rigid, coarse-grained fuzzing strategy composition, limiting the flexibility to adaptively combine the strengths of different fuzzing strategies at runtime. To address these challenges, we present xFUZZ, a flexible and extensible fuzzing framework supporting fine-grained, runtime-adaptive strategy composition. xFUZZ integrates popular input scheduling and mutation scheduling strategies as fine-grained, independently switchable plugins, allowing users to adaptively replace any plugins throughout the fuzzing campaign. Furthermore, we introduce an adaptive algorithm based on Sliding-Window Thompson Sampling, which dynamically selects the optimal composition of the fuzzing strategy during the fuzzing campaign. Experimental results show that xFUZZ outperforms state-of-the-art fuzzers by achieving a 10.07% increase in unique vulnerability discovery and a 4.94% improvement in code coverage. Notably, xFUZZ is the first to detect 21 out of 37 vulnerabilities in the test suite, establishing its effectiveness across varied targets.

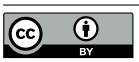
CCS Concepts: • **Software and its engineering → Software testing and debugging**; **Software testing and debugging**; • **Security and privacy → Vulnerability scanners**; **Vulnerability scanners**.

Additional Key Words and Phrases: Fuzzing, Vulnerability Detection, Software Security

ACM Reference Format:
Dongsong Yu, Yiyi Wang, Chao Zhang, Yang Lan, Zhiyuan Jiang, Shuitao Gan, Zheyu Ma, and Wende Tan. 2025. xFUZZ: A Flexible Framework for Fine-Grained, Runtime-Adaptive Fuzzing Strategy Composition. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA004 (July 2025), 23 pages. https://doi.org/10.1145/3728873

[*]Corresponding author.
[†]Also affiliated with JCSS, Tsinghua University (INSC) - Science City (Guangzhou) Digital Technology Group Co., Ltd.

Authors' Contact Information: Dongsong Yu, Zhongguancun Laboratory, Beijing, China, yudongsong@zgclab.edu.cn; Yiyi Wang, Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China and Huazhong University of Science and Technology, Wuhan, China, ahuo2865189826@gmail.com; Chao Zhang, Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China and Zhongguancun Laboratory, Beijing, China, chaoz@tsinghua.edu.cn; Yang Lan, Zhongguancun Laboratory, Beijing, China, lanyang0908@gmail.com; Zhiyuan Jiang, National University of Defense Technology, Changsha, China, jzy@nudt.edu.cn; Shuitao Gan, Laboratory for Advanced Computing and Intelligence Engineering, Zhengzhou, China, ganshuitao@gmail.com; Zheyu Ma, Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China, mzy20@mails.tsinghua.edu.cn; Wende Tan, Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China, twd2.me@gmail.com.



ACM 2994-970X/2025/7-ARTISSTA004
https://doi.org/10.1145/3728873

1 Introduction

Fuzzing has become an indispensable tool for software vulnerability detection, particularly through automatically generating or mutating test cases and feeding them to programs under test (PUTs) to trigger potential security violations. Coverage-based Grey-box Fuzzing (CGF) is a widely adopted fuzzing approach that enhances vulnerability detection by using lightweight instrumentation to guide fuzzing process and increase code coverage. While numerous CGF enhancements have been proposed — targeting areas like input scheduling strategies [6, 7, 32, 40, 43] and mutation scheduling strategies [19–21, 24, 41, 46] — these optimizations often address specific aspects in isolation. Evaluations on standardized benchmarks reveal that these improvements are inconsistently effective across different programs, due to the characteristics of each target. Combining multiple optimizations in a single framework could address this variability, yet existing approaches lack the fine-grained adaptability needed to dynamically adjust strategies in response to runtime feedback.

Existing collaborative fuzzing techniques [9, 14, 16, 28] attempt to improve fuzzing by coordinating multiple tools but are limited by their coarse-grained strategy switching. These approaches can adjust strategy only at the tool level, assigning resources between different fuzzer tools rather than dynamically adapting specific components within each tool. For instance, existing collaborative fuzzing solutions cannot apply AFLFast's input scheduling strategy [7] together with MOpt's mutation scheduling strategy [24] at runtime. Instead, they can only run AFLFast and MOpt independently, which increases fuzzing costs and misses the potential synergies between strategies. The limitation constrains the flexibility of collaborative fuzzing efforts, as they cannot fully leverage the diverse strengths of different fuzzing strategies to achieve optimal performance.

In contrast, frameworks like LibAFL [13] offer modular architectures, with individual fuzzing strategies designed as separate components. While this decoupling allows for initial customization, it lacks the runtime adaptability essential for optimal fuzzing. Importantly, the most effective strategy combination is not fixed: optimal fuzzing strategies can vary significantly across different targets, and even for a single target, the best strategy may shift over time as fuzzing progresses. Since these solutions set their strategy configurations at the beginning of a fuzzing campaign, they struggle to achieve optimal results. Without runtime adjustments, it is challenging to identify the most suitable strategies for a target beforehand, and the static configurations cannot adapt to evolving conditions during execution.

To this end, we propose xFUZZ, a novel fuzzing framework that combines fine-grained component control with runtime-adaptive strategy composition. xFUZZ is characterized by granular switching of fuzzing components at runtime, featuring popular input scheduling and mutation scheduling strategies as modular plugins. Each plugin functions independently and can be replaced during fuzzing, giving xFUZZ unprecedented flexibility to fine-tune configurations based on target-specific requirements at runtime. The fine-grained control allows xFUZZ to continuously optimize its fuzzing strategy composition without pausing or restarting the fuzzing campaign. Furthermore, xFUZZ introduces a Thompson Sampling-based runtime adaptation algorithm to facilitate automatic strategy composition based on real-time performance. The algorithm dynamically selects and adjusts the most effective fuzzing strategies throughout the fuzzing process, which ensures that the fuzzing campaign continuously adapts its configuration to optimize vulnerability detection. Evaluation based on public benchmarks shows that xFUZZ outperforms state-of-the-art fuzzing tools by discovering 10.07% more unique vulnerabilities. In addition, xFUZZ was the first to detect 21 out of the 37 vulnerabilities in the test suite. In terms of code coverage, xFUZZ achieved 4.94% higher coverage than the latest version of AFL++ and 11.48% more than autofz [14].

- We present xFUZZ, a fuzzing framework written from scratch, that enables fine-grained, runtime-adaptive control over fuzzing strategies, addressing the limitations of both coarse-grained collaborative fuzzing and static modular fuzzing tools.
- We propose a customized Thompson Sampling algorithm that leverages xFUZZ to dynamically select the optimal fuzzing strategy composition based on real-time feedback.
- We demonstrate superior experimental results, with xFUZZ discovering more unique vulnerabilities in less time and achieving higher code coverage compared to state-of-the-art tools.
- We will open-source the xFUZZ framework to foster further research and development, providing a flexible foundation for extending and enhancing fuzzing strategies within the fine-grained and runtime-adaptive architecture.

2 Background & Related Work

2.1 Fuzzing

Fuzzing is a dynamic software testing technique for vulnerability detection, typically divided into white-box, black-box, and grey-box approaches based on access to internal program details [15, 22]. White-box fuzzing leverages internal structures like control flow graphs [2, 29], black-box fuzzing generates random inputs without program knowledge [26], and grey-box fuzzing uses lightweight runtime feedback such as code coverage [12, 13, 31, 45]. Fuzzing is also classified by input generation: mutation-based, which modifies existing inputs [22, 33], and generation-based, which creates inputs from models or specifications [5, 11, 27, 30, 34, 38].

2.2 Fuzzing Strategy

Over the past decades, numerous enhancements have been introduced to improve the scheduling of different elements within the fuzzing pipeline. These strategies generally focus on input scheduling and mutation scheduling to optimize fuzzing efficiency.

1) **Input scheduling strategy**: Input scheduling includes both power scheduling and seed selection. In coverage-based greybox fuzzing, power scheduling assigns different "energy" levels to seeds, controlling how many resources a seed will consume in the mutation phase. Notable approaches like AFLFast [7] introduce energy allocation algorithms—such as *explore*, *exploit*, *coe*, *fast*, *lin*, and quad—based on factors like coverage density and edge frequency. Other methods, such as the entropy-based power scheduling in Entropic [6], also attempt to prioritize seeds likely to yield novel coverage. Seed selection strategies, on the other hand, focus on prioritizing seeds that have the potential to reveal new paths or trigger crashes. For example, Angora [8] enhances seed selection by incorporating call stacks, while AFL-Sensitive [39] uses memory-access-aware branch coverage and n-gram branch coverage to improve precision.
2) **Mutation scheduling strategy**: Most fuzzers include multiple mutators (e.g., bitflip, arithmetics,*et al.*) and often apply several of them in combination on a single seed to generate higher-quality inputs, as seen in the havoc stage of AFL. In the havoc stage, the choice of mutator is typically random. The core idea behind mutator scheduling is to optimize the selection probability of mutators, enabling the fuzzer to more effectively choose the best-performing mutators and, in turn, generate higher-quality test cases. MOpt [24] is the first fuzzing solution to propose this idea, using a customized Particle Swarm Optimization (PSO) algorithm based on the historical performance of different mutators to optimize mutator selection. Wu *et al.* [41] uses the Upper Confidence Bound (UCB) algorithm to simultaneously optimize the number of mutations and mutator selection during the havoc stage. Seamfuzz [20], on the

other hand, groups different seeds and uses Thompson Sampling to optimize the mutator selection for each group.

Many fuzzing strategies improve specific aspects of fuzzing but are often implemented by forking tools like AFL, leading to a fragmented ecosystem. Merging these forks demands substantial engineering, hindering integration and fair comparison.

AFL++ [12] mitigates this by unifying advanced fuzzing techniques in a community-driven framework, easing the extension and testing of new mutators. However, AFL++ does not fully decouple its integrated strategies, which limits the flexibility needed to combine techniques orthogonally. The tight integration restricts modularity, making it difficult for researchers to experiment with and adopt new approaches in a truly combinable and adaptable manner.

2.3 Extensible Fuzzing Framework

To fully leverage the strengths of various fuzzing strategies, Fioraldi *et al.* developed LibAFL [13], a fully extensible fuzzing framework that defines nine core components common to modern fuzzers in Rust. LibAFL allows researchers to independently implement and combine strategies across these components, enabling tailored fuzzer configurations and facilitating fair comparisons of individual techniques. The modularity helps mitigate the integration challenges faced by other fuzzing tools.

Although LibAFL provides a modular framework that allows for the orthogonal combination of different fuzzing strategies, it does not support real-time strategy adaptation. LibAFL was designed primarily for building fuzzers, meaning that all configurations are fixed at compile-time. Once compiled, the fuzzer operates with a fixed set of strategies, preventing dynamic component replacement during fuzzing.

Furthermore, we illustrate that LibAFL's implementation lacks sufficient component decoupling, as it was not initially designed with runtime strategy adaptation in mind. For example, when writing a fuzzer, it is possible to specify the mutation strategy by configuring parameters of the *stage* trait and to define the input scheduler through the *scheduler* trait defined by LibAFL. However, once the compilation is complete, both *stage* and *scheduler* are embedded as arguments in the implementation of the *fuzzer* trait, preventing their independent modification in memory.

2.4 Collaborative Fuzzing

To leverage the strengths of multiple fuzzers at runtime, collaborative fuzzing approaches [9, 14, 16, 28] have been developed to run multiple fuzzers simultaneously, sharing seeds through a global pool. In resource-constrained environments, they dynamically allocate more resources to higher-performing fuzzers, thereby enhancing overall fuzzing effectiveness.

However, the collaborative fuzzing approach to the composition of the fuzzing strategy has two main limitations. Take autofz [14], the state-of-the-art collaborative fuzzer, as an example. First, it treats entire fuzzers as composition units, preventing fine-grained orthogonal combinations of strategies from different tools. Second, autofz's performance is constrained by engineering differences among the fuzzers it uses. For example, optimizations such as persistent mode and shared memory are not supported by every tool, limiting the ability of autofz to fully maximize resource efficiency.

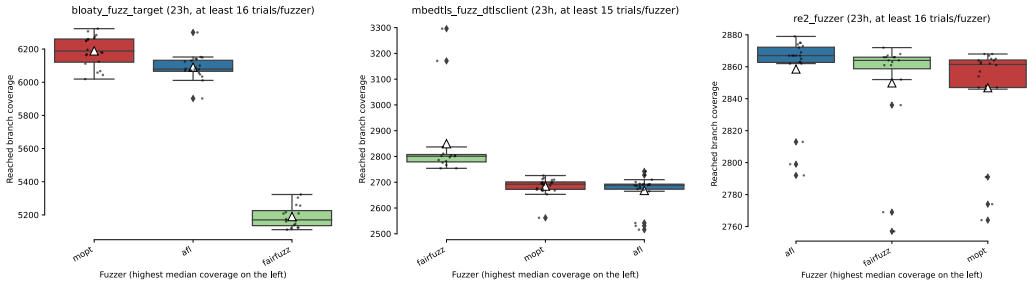
2.5 Comparision of Existing Solutions

Table 1 compares several mainstream fuzzing solutions, highlighting differences in strategy composition, extensibility, runtime adaptability, and strategy hot-swapping capabilities.

- **Strategy Composition Unit**: AFL forks and AFL++ use a monolithic architecture, with tightly coupled components that restrict flexible composition. autofz operates at the fuzzer

Table 1. Comparison of Existing Fuzzing Solutions

	Strategy Composition Unit	Extensibility	Runtime Adaptability	Strategy Hot-Swapping
AFL forks	monolithic	✗	✗	✗
AFL++	monolithic	mutator only	✗	✗
LibAFL	component-level	component-level	✗	✗
autofz	fuzzer-level	fuzzer extension	fuzzer-level	✗
xFUZZ	component-level	component-level	component-level	✓



(a) Comparison of several mutation scheduling strategies on *bloaty*

(b) Comparison of several mutation scheduling strategies on *mbedtls*

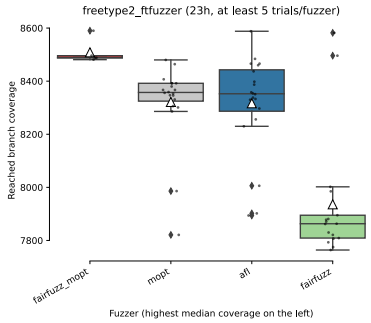
(c) Comparison of several mutation scheduling strategies on *re2_fuzzer*

Fig. 1. Reached branch coverage comparison of several mutation scheduling strategies

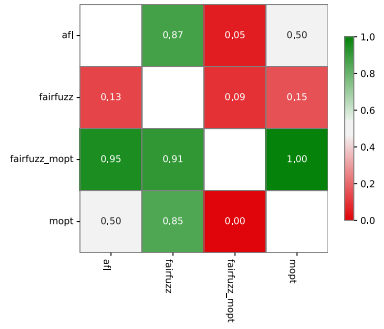
level, orchestrating multiple fuzzers to increase code coverage, but lacks fine-grained control over individual strategies. In contrast, LibAFL and xFUZZ adopt a component-level approach, allowing more granular customization of individual fuzzing strategies.

- **Extensibility**: While AFL forks provide no extensibility, AFL++ allows only mutator extensions, enabling limited customization. LibAFL and xFUZZ, however, provide component-level extensibility, supporting the addition of new fuzzing strategies and algorithms across multiple components, allowing researchers to easily experiment with and integrate different techniques.
- **Runtime Adaptability**: AFL forks, AFL++, and LibAFL do not support runtime adaptations, restricting flexibility during fuzzing. autofz allows fuzzer-level runtime control, dynamically adjusting resource allocation among fuzzers. xFUZZ stands out by enabling component-level runtime adaptability, allowing it to reconfigure individual components dynamically to better adapt to target-specific and phase-specific needs during fuzzing.
- **Strategy Hot-Swapping**: xFUZZ uniquely supports hot-swapping of fuzzing strategies at the component level, allowing users to change specific strategies without restarting the fuzzing process. This feature provides significant flexibility, enabling dynamic adjustments to strategies in response to runtime performance data, a capability not available in any of the other compared fuzzing solutions.

In general, the combination of features makes xFUZZ particularly advantageous for users who require fine-grained control and extensibility in their fuzzing runtime.



(a) Reached branch coverage comparison on *freetype2*



(b) The table summarizes the A12 values from the pairwise Vargha-Delaney A measure of effect size. Green cells indicate the probability that the fuzzer in the row will outperform the fuzzer in the column.

Fig. 2. Comparison of mutator scheduling strategies on *freetype2*. The fairfuzz_mopt configuration utilized FairFuzz as the mutation scheduling strategy during the first half of the testing period, followed by MOpt in the latter half.

3 Motivation Examples

In this section, we analyze the results from multiple fuzzing strategies to illustrate their variability in performance and highlight the need for runtime-adaptive fuzzing strategy composition.

Figure 1 shows the fuzzbench code coverage report of three mutation scheduling strategies adopted by AFL (havoc), Fairfuzz [21], and MOpt [24]. Each fuzzer was tested with 24-hour experiments, and the report shows the reached branch coverage distribution on different programs. For example, while MOpt achieves the highest median coverage on *bloaty*, Fairfuzz performs better on *mbedtls*, and AFL (havoc) is the most efficient mutation scheduling strategy on *re2_fuzzer*.

In addition, prior research [13] has shown that the coverage performance of AFLFast's input scheduling strategies (such as *fast*, *coe*, and *explore*) also varies significantly across different targets. The variability also underscores the limitations of fuzzer-level approaches, which can only allocate resources to AFLFast as a whole but cannot adjust its fine-grained input scheduling strategies. Consequently, fine-grained control over these strategies is essential, as it would allow for dynamic selection and adaptation of the most effective input scheduler or mutation scheduler based on the target's specific characteristics. The example highlights that a one-size-fits-all approach is insufficient for optimizing fuzzing effectiveness, and fine-grained adjustments are crucial for adapting to the unique demands of each target.

Observation 1: The optimal fuzzing strategy is target-dependent.

Furthermore, we conducted a simple experiment that combined the FairFuzz and MOpt mutation scheduling strategies on *freetype2*. In a 24-hour experiment, we applied the FairFuzz mutation scheduling strategy for the first 12 hours and then switched to the MOpt mutation scheduling strategy for the remaining 12 hours. Figure 2 shows the comparison of the mutator scheduling strategies. The results in terms of code coverage and A_{12} values indicate that this naive runtime strategy switching achieves a better result. The results indicate that a simple runtime switch between strategies can achieve higher code coverage than relying on any single mutation scheduling strategy alone. Moreover, even strategies that appear optimal individually may not maintain their

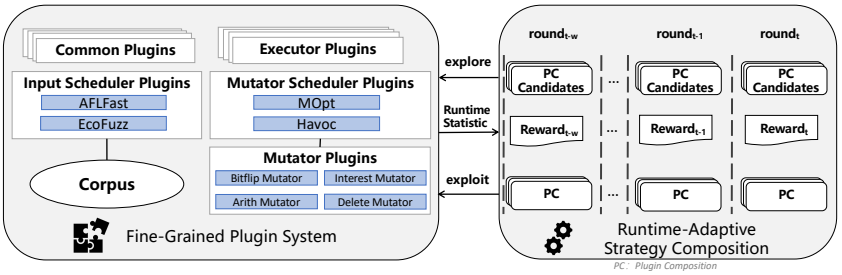


Fig. 3. Overall architecture of the xFUZZ

effectiveness throughout the fuzzing session. This experiment demonstrates that switching to a different strategy at a specific point can yield better results.

Observation 2: The fuzzing strategy that appears most effective at one stage of the fuzzing process may not remain optimal throughout all stages of fuzzing.

Based on the above observations, the optimal fuzzing strategy should adapt dynamically, as the most effective configuration can vary significantly with both the target programs and the different stages of fuzzing over time. It calls for a flexible and extensible fuzzing framework that supports fine-grained, runtime strategy adjustments. The framework would allow strategies to be selected and modified at runtime, optimizing fuzzing performance by continuously adapting to each target's unique characteristics and evolving needs throughout the fuzzing process.

4 Methodology

4.1 Overview of xFUZZ

The primary goal of the xFUZZ framework is to enable fine-grained, runtime-adaptive control over fuzzing strategies. The architecture of xFUZZ is shown in Figure 3.

To achieve fine-grained control over fuzzing strategies, xFUZZ leverages a plugin-based architecture, where each key component of the fuzzing pipeline-such as input scheduling, mutation scheduling, mutation, and execution-is implemented as an independent plugin. The design enables each plugin to operate autonomously as a shared library, allowing it to be loaded or unloaded at runtime without interrupting the fuzzing process. The modularity facilitates the runtime assembly of optimized strategy compositions tailored to the requirements of each target.

Furthermore, to enable runtime-adaptive fuzzing strategy composition, we use Sliding-Window Thompson Sampling (SW-TS) to determine the current most effective fuzzing strategy composition based on recent performance. Specifically, xFUZZ uses a two-phase process of explore and exploit. During the exploration phase, it runs multiple plugin composition (PC) candidates to gather runtime statistics of different fuzzing strategy compositions. Each candidate is then assigned a reward based on these statistics, reflecting the effectiveness of different compositions. In the exploitation phase, the best-performing composition is selected according to the SW-TS approach. This cyclical process continuously refines the fuzzing strategy, enabling xFUZZ to dynamically adjust to evolving conditions as fuzzing progresses.

4.2 Fine-Grained Plugin System

xFUZZ decomposes the fuzzing workflow into several key components, each implemented as a distinct plugin category. Each category is orthogonal to the others, allowing flexible compositions of plugins that implement different algorithms across categories. To support different types of

fuzzing strategies at a granular level and ensure the integrity of these strategies while allowing them to be orthogonal, xFUZZ has defined the following categories of plugins related to fuzzing strategies:

- **Input Scheduler Plugins**: Input Scheduler manages the prioritization and energy allocation for each seed, determining which seeds should be mutated and how much computational resource each one should receive. Input Schedulers like AFLFast and EcoFuzz are implemented as interchangeable plugins, allowing xFUZZ to select the most effective scheduling strategy dynamically.
- **Mutator Scheduler Plugins**: The Mutator Scheduler is responsible for managing the entire mutation process within the seed mutation task. It uses the mutator as the core scheduling unit, guiding the selection of mutation operators and mutation locations. Currently, the scheduler primarily supports two strategies: random mutation (Havoc) and mutation based on the Particle Swarm Optimization algorithm (MOpt).
- **Mutator Plugins**: Mutator implements various common mutation operations in fuzzing, including bit flipping, addition/subtraction, and dictionary mutators, *et al.* Within the xFUZZ framework, various mutators can be freely chosen and scheduled by the Mutator Scheduler without interrupting the fuzzing process.

Each plugin is designed as a dynamic shared library managed via *dlopen*, *dlsym*, and *dlclose*, allowing xFUZZ to load, initialize, or unload plugins based on runtime-adaptive composition.

Furthermore, to ensure continuous and effective decision-making throughout the fuzzing process, it is essential to preserve and restore the entire state information of the fuzzing process during plugin switching. By maintaining the integrity of fuzzing statistics, seed state, and plugin state information, xFUZZ can seamlessly transition between different fuzzing strategies without losing valuable historical data. The capability is particularly crucial for evolutionary algorithms that depend on past performance metrics—such as path frequency in AFLFast and mutator efficiency in MOpt—to optimize future fuzzing actions.

In summary, xFUZZ 's runtime plugin switching mechanism enables fine-grained control over fuzzing strategies while preserving critical state data, ensuring consistent performance and maximizing the overall effectiveness throughout the fuzzing campaign. Unlike switching an entire fuzzer, xFUZZ 's approach targets individual plugins (i.e., distinct fuzzing strategies), maintaining global and seed state information without disruption. The preservation of essential data is crucial for continuous, informed decision-making during fuzzing, particularly for strategies that rely on accumulated performance metrics to optimize future actions.

4.3 Runtime-Adaptive Fuzzing Strategy Composition

4.3.1 Sliding-Window Thompson Sampling. Due to the diversity of strategy compositions and the uncertainty in the performance of different compositions during execution, xFUZZ models the runtime-adaptive control over fuzzing strategies as a Multi-Armed Bandit (MAB) problem. While MAB has been widely adopted in fuzzing for tasks such as input scheduler [43] and mutator scheduler [20, 24], the primary challenge in our scenario is the non-stationary nature of the reward distributions. In the traditional MAB approaches [4, 10, 36], each choice (arm) has a fixed reward distribution, and the goal is to maximize rewards by balancing exploration (trying different arms) and exploitation (selecting the best-known arm). However, as observed in Section 3, the optimal strategy tends to change over time as the program space expands and new seeds are added to the corpus. This behavior aligns more closely with the characteristics of Non-Stationary MAB (NS-MAB) problems. Further evaluation in section 6.5 shows that the Sliding-Window Thompson

Sampling used for NS-MAB significantly outperforms the classic Thompson Sampling for traditional MAB on certain targets.

To address this issue, we employ the Sliding-Window Thompson Sampling (SW-TS) alogorithm [37]. Unlike traditional MAB methods that consider all past observations to estimate the probability distributions of each arm, SW-TS focuses only on the most recent observations within a defined sliding window. This approach allows xFUZZ to more effectively adapt to changing conditions during fuzzing, making it better suited to the dynamic and evolving nature of fuzzing campaigns, where optimal strategies may shift as new inputs are explored.

By using SW-TS, we prioritize more recent fuzzing statistics within specific rounds (or windows) of exploration and exploitation. This enables xFUZZ to dynamically adjust its plugin selection based on the latest feedback, ensuring that the fuzzing strategy remains aligned with current conditions. Specifically, SW-TS is designed to address NS-MAB by setting a sliding window value, allowing outdated reward information to be effectively discarded, and ensuring that the fuzzing process remains agile and responsive to changes in the environment. In the t-th round, for plugin composition i, the sampling value $\hat{\theta}_{i,t}$ is calculated as follows:

$$\hat{\theta}_{i,t} \sim \text{Beta}(R^P_{i,t}, R^N_{i,t}) \qquad R^P_{i,t} := \sum_{j=\max(1,t-w)}^{t} r^P_{i,j} \qquad R^N_{i,t} := \sum_{j=\max(1,t-w)}^{t} r^N_{i,j}$$

where Beta represents the Beta distribution function, w is the sliding window value, $r^P_{i,j}$ is the positive reward for plugin composition i in the j-th round, and $r^N_{i,j}$ is the negative reward for plugin composition i in the j-th round. Detailed implementation and steps of this algorithm are provided in Algorithm 1.

Algorithm 1 Sliding-Window Thompson Sampling

```
Input:
    ℙℂ ← plugin composition to be sampled
    round ← current round
Output:
    sample ← the sampling value of plugin composition
procedure SW_THOMPSON_SAMPLING(ℙℂ, round)
    Reward^P ← 0, Reward^N ← 0
    for i = max(1, round − sw_value) to round do        ▷ sw_value is the sliding window value
        Reward^P ← Reward^P + reward^P_{PC,i}
        Reward^N ← Reward^N + reward^N_{PC,i}
    end for
    sample ← Beta(Reward^P, Reward^N)
    return sample
end procedure
```

4.3.2 The Reward Mechanism for SW-TS. In the SW-TS algorithm, it's important to clearly define what constitutes a "positive reward" (a successful or desirable outcome) and a "negative reward" (an unsuccessful or undesirable outcome) for each arm. During the fuzzing process, the most intuitive evaluation criterion is based on the number of new paths (i.e., new seeds), where the number of new seeds discovered by a specific plugin composition over a period is considered a positive reward, while the number of test cases that did not discover new paths is considered a negative reward. However, throughout the fuzzing process, the discovery of new paths is typically rare, especially in the late stage of fuzzing, which results in positive rewards being significantly lower than negative

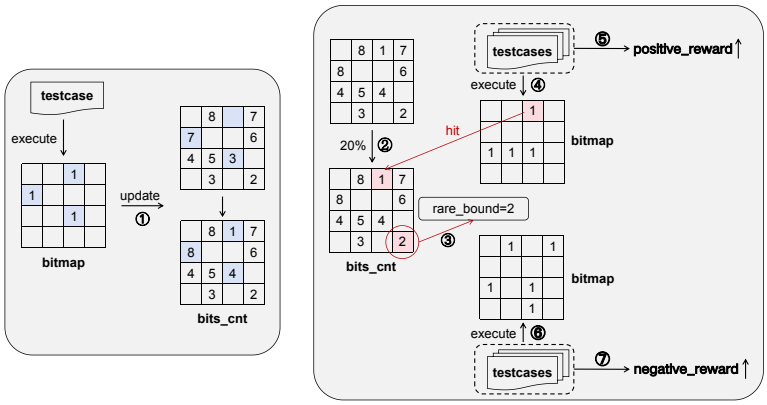


Fig. 4. Illustration of reward mechanism

rewards, reducing the distinction between different arms during sampling and making it difficult to effectively differentiate the best plugin composition.

To improve the distinction between different strategy compositions in the SW-TS algorithm, it is essential to lower the threshold for positive rewards, thereby increasing the number of positive rewards. To achieve this, we introduce a more sensitive metric for defining the reward mechanism, as illustrated in Figure 4. Throughout the fuzzing process, we monitor the bitmap information of each executed testcase. For every bit set to 1 in the bitmap, we increment the corresponding entry in a global statistics array, **bits_cnt**, to track the frequency of each bit's occurrence (①).

At the beginning of each fuzzing round, we utilize the frequencies of the bits in **bits_cnt** to determine a threshold, referred to as **rare_bound** (②-③). During this round, if a mutated input's bitmap hits any bit with a frequency below **rare_bound** (including bits with a frequency of 0, indicating they have not been hit previously), we classify this as a "hit" on rare bits, and it is deemed a positive reward. Consequently, the global variable **positive_reward** is incremented (④-⑤). On the other hand, if no such bits are hit, the input is classified as a negative reward, and the variable **negative_reward** is incremented (⑥-⑦). In the current implementation, **rare_bound** is defined as the 20th percentile of the lowest frequencies. This threshold increases the likelihood of successfully hitting rare bits, while effectively preventing an excessive number of positive rewards.

Note that xFUZZ uses AFL-type bitmap to store the coverage, where different bytes represent different branches of the program, and different bits within the same byte represent different hit counts for the same branch. Thus, this reward mechanism encourages xFUZZ to favor plugin compositions that generate more test cases hitting rare branches. At the same time, when multiple plugin compositions hit different rare branches, the mechanism prevents xFUZZ from staying too long on a single plugin composition. Once a plugin composition is selected, the rare branches it hits will gradually become "less rare," thus providing more opportunities for other compositions.

4.3.3 Tiered Exploration Strategy. In xFUZZ, the composition of plugins exhibits multiplicative characteristics. For example, if there are three different plugins in each category and there are three categories, there will be a total of 27 (3*3*3) different compositions. Attempting to evaluate all these compositions during the exploration will consume significant resources. We will demonstrate it through experiments in Section 6.4. Therefore, we have adopted a tiered exploration strategy aimed

at reducing the required exploration time while ensuring an effective approximation to the global optimum. The detailed process of this strategy is shown in Algorithm 2.

Algorithm 2 Tiered Exploration Strategy

```
1: Input:
2:     ℂ ← {C_1, C_2, ..., C_n}                          ▷ C_i is the set of plugins in i-th category
3: Output:
4:     𝕊 ← {sp_1, sp_2, ..., sp_n}                       ▷ sp_i is the selected plugin from C_i
5: procedure TIERED_EXPLORATION(ℂ, round)
6:     for i = 1 to n do
7:         𝕊[i] ← sp_i_pre
8:     end for
9:     for i = 1 to n do
10:        sp_i_selected ← SINGLE_EXPLORATION(i, C_i, 𝕊, round)
11:        𝕊[i] ← sp_i_selected
12:        sp_i_pre ← sp_i_selected                      ▷ Store the results of current round
13:    end for
14:    return 𝕊
15: end procedure
16: ──────────────────────────────────────────────────────────────────────
17: Input:
18:    C_i ← {P_1, P_2, ..., P_m}                        ▷ P_j is the j-th plugin in C_i
19:    𝕊 ← {sp_1, sp_2, ..., sp_n}                       ▷ sp_i is the selected plugin from C_i
20: Output:
21:    sp_i_selected                                     ▷ Best plugin in C_i via SW-TS
22: procedure SINGLE_EXPLORATION(i, C_i, 𝕊, round)
23:    for j = 1 to m do
24:        𝕊_tmp ← 𝕊
25:        𝕊_tmp[i] ← P_j
26:        (Reward^P, Reward^N) ← RUN_XFUZZ(𝕊_tmp)
27:        RECORD_REWARD(𝕊_tmp, (Reward^P, Reward^N), round)   ▷ Record the reward of plugin composition
   𝕊_tmp in round round
28:        sample_j ← SW_THOMPTON_SAMPLING(𝕊_tmp, round)
29:    end for
30:    max_sample ← −1
31:    for j = 1 to m do
32:        if max_sample < sample_j then
33:            max_sample ← sample_j
34:            sp_i_selected ← P_j
35:        end if
36:    end for
37:    return sp_i_selected
38: end procedure
```

Assume that xFUZZ consists of n categories of plugins. In each exploration round, xFUZZ gradually selects the best-performing plugin from each plugin category, fixes the explored categories with their optimal plugins, and continues to explore the next plugin category (Line 9-13). For every category of plugins, xFUZZ tries different plugin options and obtains rewards for the compositions that include the current plugin (Line 23-29). Then, based on the reward mechanism described in Section 4.3.2, the reward distribution of SW-TS is updated (Line 27), and sampling values are calculated from the rewards within the sliding window using the algorithm outlined in Algorithm 1 (Line 28). Based on these values, xFUZZ selects the optimal plugin in each category to retain in the configuration and proceeds to explore the next category of plugins (Line 31-37).

Upon completing this exploration round, the best-performing plugin composition is selected for a focused fuzzing phase (i.e., exploitation), which is executed for a designated period. The rewards during the exploitation phase are also used to update the reward distribution of SW-TS. The settings for the exploitation time will be discussed in Section 6.3.

To enable xFUZZ to capture the dynamic behavior during the fuzzing phase quickly, the test duration for each selected plugin is set to 1 minute, accelerating the exploration process (Line 26). This allows xFUZZ to rapidly iterate through multiple rounds of exploration, dynamically adjusting its composition in real-time. Additionally, at the beginning of each round, all plugin category selections are initialized based on the selected plugin composition from the previous round, thereby leveraging the exploration results from the prior iteration (Line 6-8). For the initial exploration round, we use "explore mode" as the input scheduler and "havoc" as the mutator scheduler as the initial plugin configuration. This configuration is the default mode of AFL++ and has been identified as the optimal setup through multiple evaluations.

With the tiered exploration, we sequentially optimize each category of plugins rather than exhaustively exploring all compositions at once. It transforms the complex multiplicative composition exploration into a simpler additive exploration, significantly reducing the time expenditure of the exploration phase while also ensuring an effective pursuit of the global optimum.

5 Implementation

We developed the xFUZZ framework using C++, leveraging class polymorphism to accommodate different implementations of the same type of strategy plugins. xFUZZ's strategy plugins are designed as dynamic libraries, utilizing *dlopen* and *dlclose* for loading and unloading plugins. Additionally, we implemented the xFUZZ Manager to provide external interaction capabilities via JsonRPC.

xFUZZ comprises approximately 75k lines of C++ code, including around 9k lines for the core framework and 66k lines for plugin implementations. Additionally, we implement xFUZZ's runtime-adaptive fuzzing strategy composition algorithm using approximately 800 lines of Python to interact with the plugin system.

This paper discusses only the plugins corresponding to the following fuzzing strategies (as shown in Table 2). Overall, these plugins offer 4 options for the Input Scheduler and 2 for the Mutator Scheduler, along with an additional extension to the Mutator Scheduler called Splice. Splice is an AFL feature that, in addition to regular mutations, splices two seeds before applying mutations. We have integrated Splice as an extension to the Mutator Scheduler in xFUZZ, allowing it to freely combine with any Mutator Scheduler. However, due to its potential impact on the algorithmic data integrity of the EcoFuzz plugin, Splice cannot be used in combination with EcoFuzz. Therefore, xFUZZ offers a total of **14** (4*2*2-2) different compositions. In addition to the plugins mentioned above, there are 30 mutator plugins available for the Mutator scheduler, implementing mutation operators such as bitflip, dictionary, and arithmetic. All mutators are bit-level mutators, scheduled by the Havoc and MOpt mutator schedulers, and are enabled in the experiments presented in this paper. At fuzzing startup or when a strategy composition switch is required, the corresponding plugins are provided to xFUZZ in JSON format, enabling the xFUZZ plugin system to load the specified composition of plugins.

6 Experiments

In this section, we evaluate xFUZZ to answer the following research questions:

- **RQ1: Code Coverage Capability of xFUZZ**: How does the code coverage capability of xFUZZ compare to both individual and collaborative fuzzing approaches?

Table 2. xFUZZ Strategy Plugins and Implementations

Strategies	Implemented as	Plugin Categories
AFLFast Fast Mode[7, 12]	Parameters of the AFLFast plugin	Input Scheduler
AFLFast Explore Mode[7, 12]	Parameters of the AFLFast plugin	Input Scheduler
AFLFast Coe Mode[7, 12]	Parameters of the AFLFast plugin	Input Scheduler
EcoFuzz[43]	EcoFuzz Plugin	Input Scheduler
Havoc[12, 45]	Havoc Plugin	Mutator Scheduler
MOpt[12, 24]	MOpt Plugin	Mutator Scheduler
Splice[45]	Extension Plugin	Extension to Mutator Scheduler

- **RQ2: Vulnerability Discovery Ability of xFUZZ**: How does xFUZZ's ability to discover vulnerabilities in terms of both discovery speed and quantity?
- **RQ3: Exploit Time Setting for xFUZZ**: What is the optimal exploit time setting for xFUZZ during the exploitation phase to maximize overall performance?
- **RQ4: Effectiveness of xFUZZ 's Tiered Exploration Strategy**: How does xfuzz's tiered exploration strategy perform compared to the global exploration strategy?
- **RQ5: Effectiveness of xFUZZ's SW-TS Algorithm**: How effectively does the SW-TS Algorithm enhance xFUZZ's ability to make optimal fuzzing strategy compositions and improve its overall performance?
- **RQ6: Effectiveness of xFUZZ's Runtime-Adaptive Control**: How does xFUZZ's runtime-adaptive control impact the fuzzing process compared to its static configuration?

All evaluation experiments were conducted on a machine running Ubuntu 20.04 with 128 cores and 1.8TB of memory, powered by an Intel(R) Xeon(R) Platinum 8358 CPU. We will address these six questions in Sections 6.1 to 6.6.

6.1 Code Coverage Capability of xFUZZ

To evaluate the code coverage capability of xFUZZ, we conducted comparative experiments against both individual and collaborative fuzzing approaches.

6.1.1 Comparison with Individual Fuzzing. We compared xFUZZ with five baseline individual fuzzers (AFL++ 4.21a, AFL 2.57b, AFLFast, MOpt, EcoFuzz) using Fuzzbench [25]. The experiments were set to run for 24 hours and were repeated 10 times. The 12 target programs used for these experiments are all from Fuzzbench.

As shown in Table 3, we calculated the average code coverage of six fuzzers across 12 target programs. The Mann-Whitney U test and Vargha-Delaney $\hat{A}$ measure highlight statistically significant differences across the statistics. In terms of average coverage, xFUZZ achieved the highest average coverage on 9 target programs and exceeded the second-ranked AFL++ by 4.94% across all 12 target programs. Additionally, xFUZZ demonstrated statistically significant improvements over AFL++ on 8 of the target programs. On *file*, *lcms* and *mbedtls* in particular, xFUZZ's average coverage exceeds that of AFL++ by more than 20%.

6.1.2 Comparison with Collaborative Fuzzing. We selected the state-of-the-art collaborative fuzzer, autofz[14], for evaluation and compared its performance with xFUZZ. Since the autofz open-source project[35] does not provide detailed build information and cannot be built on Fuzzbench, we used the docker image provided by autofz for our comparison. This evaluation was conducted from 12 target programs in Unifuzz[23]. In this experiment, autofz was configured with its optimal settings as described in its paper, utilizing 10 integrated fuzzers (AFL[45], AFLFast[7], MOpt[24],

Table 3. Branch Coverage Results for 6 Fuzzers Across 12 Target Programs in Fuzzbench, Over 10 Runs of 24 Hours Each

Program	xFUZZ	AFL++			AFL		
	Cover	**Cover**	**R$_{cov}$**	**p value/$\hat{A}_{12}$**	**Cover**	**R$_{cov}$**	**p value/$\hat{A}_{12}$**
bloaty	**6236.3**	5986.7	+4.17%	$< 10^{-3}$/1.0	5983.6	+4.22%	$< 10^{-3}$/1.0
file	2263.7	1835.9	+23.30%	$< 10^{-3}$/1.0	**2296.3**	-1.42%	0.203/0.31
harfbuzz	**10976.2**	10949	+0.25%	0.970/0.51	10755.9	+2.05%	$< 10^{-3}$/1.0
lcms	**1649.6**	1315.9	+25.36%	0.009/0.85	1114.2	+48.05%	0.004/0.86
libjpeg	3209.3	**3211.4**	-0.07%	0.007/0.15	2544	+26.15%	$< 10^{-3}$/1.0
libpng	1987	**1995**	-0.40%	0.449/0.40	1933.1	+2.79%	0.003/0.88
mbedtls	**3286.8**	2698.7	+21.79%	$< 10^{-3}$/1.0	2643.7	+24.33%	$< 10^{-3}$/1.0
mruby	**7985.3**	7440.2	+7.33%	$< 10^{-3}$/1.0	7587.9	+5.24%	0.004/0.89
openssl	**5832.1**	5828.2	+0.07%	0.013/0.83	5824.3	+0.13%	$< 10^{-3}$/1.0
vorbis	**1268.3**	1263.8	+0.36%	0.004/0.88	1233.4	+2.83%	$< 10^{-3}$/1.0
woff2	**1131**	1126	+0.44%	0.009/0.76	1103.6	+2.48%	$< 10^{-3}$/1.0
zlib	**460.9**	458.7	+0.48%	0.182/0.68	443.6	+3.90%	0.001/0.91
Total	**46286.5**	44109.5	+4.94%	-	43463.6	+6.49%	-

Program	AFLFast			MOpt			EcoFuzz		
	Cover	**R$_{cov}$**	**p value/$\hat{A}_{12}$**	**Cover**	**R$_{cov}$**	**p value/$\hat{A}_{12}$**	**Cover**	**R$_{cov}$**	**p value/$\hat{A}_{12}$**
bloaty	5927.3	+5.21%	$< 10^{-3}$/1.0	6184.4	+0.84%	0.306/0.63	5657.8	+10.22%	$< 10^{-3}$/1.0
file	2219.0	+2.01%	0.573/0.59	2167.5	+4.44%	0.722/0.56	2004.3	+12.94%	$< 10^{-3}$/0.99
harfbuzz	10625.8	+3.30%	$< 10^{-3}$/1.0	10807.7	+1.56%	$< 10^{-3}$/1.0	10357.6	+5.97%	$< 10^{-3}$/1.0
lcms	588.3	+180.40%	$< 10^{-3}$/1.0	878.2	+87.84%	0.001/0.93	938.2	+75.83%	0.001/0.91
libjpeg	2543.9	+26.16%	$< 10^{-3}$/1.0	2544.9	+26.11%	$< 10^{-3}$/1.0	2543.3	+26.19%	$< 10^{-3}$/1.0
libpng	1936.6	+2.60%	0.001/0.93	1966.7	+1.03%	0.037/0.77	1930.8	+2.91%	0.003/0.88
mbedtls	2527.3	+30.05%	$< 10^{-3}$/1.0	2627.2	+25.11%	$< 10^{-3}$/1.0	2548.2	+28.99%	$< 10^{-3}$/0.99
mruby	7612.8	+4.89%	0.009/0.86	7702.8	+3.67%	0.002/0.91	6430.6	+24.18%	$< 10^{-3}$/1.00
openssl	5813.7	+0.32%	$< 10^{-3}$/1.0	5819.3	+0.22%	$< 10^{-3}$/1.0	5794.7	+0.65%	$< 10^{-3}$/1.0
vorbis	1236.3	+2.59%	$< 10^{-3}$/1.0	1242.6	+2.07%	$< 10^{-3}$/1.0	1230.1	+3.11%	$< 10^{-3}$/1.0
woff2	1077.4	+4.97%	$< 10^{-3}$/1.0	1103.9	+2.45%	0.001/0.97	1101.3	+2.70%	$< 10^{-3}$/0.92
zlib	444.9	+3.60%	$< 10^{-3}$/0.98	445.6	+3.43%	0.024/0.79	445.8	+3.39%	0.029/0.78
Total	42553.3	+8.77%	-	43490.8	+6.43%	-	40982.7	+12.94%	-

R$_{cov}$ represents the coverage growth rate of xFUZZ relative to the baseline fuzzer. The p-value and $\hat{A}_{12}$ measure are used to evaluate the statistical significance and effect size between xFUZZ and the baseline fuzzers.

FairFuzz[21], LearnAFL[42], QSYM[44], Angora[8], Redqueen[3], Radamsa[18], LAF-INTEL[1]). Notably, Redqueen, Radamsa, and LAF-INTEL in autofz are different configurations of AFL++[12].

Due to differences in coverage instrumentation between the two fuzzers, we used binaries instrumented with AFL++'s PCGUARD to measure coverage consistently. The experiments were conducted over 24 hours and repeated five trials with the same initial seeds from Unifuzz's dataset.

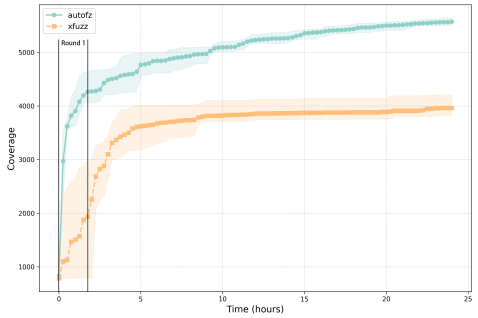
The comparison of code coverage is shown in Table 4. Among the 12 target programs, xFUZZ achieved higher average coverage than autofz in 9 programs. Notably, on *tcpdump*, *objdump*, and *nm*, xFUZZ's coverage exceeded that of autofz by 30%. It indicates that even with fewer optional strategies than autofz, xFUZZ can achieve higher coverage.

We also noticed that xFUZZ's coverage on the *exiv2* was 28.89% lower than that of autofz. To investigate this, we analyzed the coverage growth curves of autofz and xFUZZ, as well as autofz's fuzzer selection. The line chart on the left side of Figure 5 shows the coverage over time for autofz and xFUZZ on *exiv2*. It is evident that in the early stages, particularly in the first round, autofz's code coverage growth was significantly higher than xFUZZ's.

Table 4. Branch Coverage Results for xFUZZ and Autofz Across 12 Target Programs in Unifuzz, Over 5 Runs of 24 Hours Each

Program	xFUZZ	autofz		
	Cover	**Cover**	$\mathbf{R}_{cov}$	**p value**/$\hat{A}_{12}$
cflow	**1135**	1130.4	+0.41%	0.013/0.96
exiv2	3962.4	**5572.2**	-28.89%	0.008/0.0
imginfo	1810	**1819.8**	-0.54%	1.0/0.48
infotocap	1443.4	**1455.6**	-0.84%	0.69/0.6
jhead	**201**	193.2	+4.04%	0.007/1.0
jq	**1902.8**	1897.4	+0.28%	1.0/0.44
lame	**3882.6**	3705.6	+4.78%	0.012/1.0
mujs	**3383.8**	3137.2	+7.86%	0.008/1.0
nm	**2998.2**	2265.6	+32.34%	0.008/1.0
objdump	**4543.4**	2705.8	+67.91%	0.008/1.0
pdftotext	**6468.8**	6057.8	+6.78%	0.008/1.0
tcpdump	**10432**	7882.4	+32.35%	0.008/1.0
Total	**42163.4**	37823	+11.48%	-

Further analysis revealed that, in all experiments, autofz consistently selected Angora [8] during the first round. By examining the coverage growth curve of xFUZZ and autofz, we observed that autofz's coverage growth during the first round was substantially faster, a result of its early-stage selection of Angora. In autofz, each round consists of two distinct phases: The preparation phase and the focus phase. During the preparation phase, autofz rapidly explores all fuzzer options, and in the focus phase, it allocates CPU time to different fuzzers based on the findings from the preparation phase. As shown in the table on the right side of Figure 5, we summarize the code coverage after the exploration phase of the first round and the corresponding CPU time assignment for various fuzzers in autofz. It highlights that Angora was the primary choice for autofz in the first round across all repeated experiments, outperforming the other nine integrated fuzzers. The choice enabled autofz to generate a substantial number of high-quality seeds, providing it with a considerable advantage from the outset of the fuzzing process. We believe that the lower performance of xFUZZ on *exiv2* may be due to the absence of Angora. We will verify it in Appendix A.



Fuzzer	Preparation Phase of Round 1	
	Coverage	CPU Assign
AFL	825	0
AFLFast	825	0
Angora	911	1
FairFuzz	825	0
LAF-INTEL	823	0
LearnAFL	826	0
MOpt	825	0
QSYM	823	0
Radamsa	823	0
Redqueen	823	0

Fig. 5. The line chart illustrates the coverage over time for xFUZZ and autofz on *exiv2*. The table shows the coverage on *exiv2* and CPU time assignment for different fuzzers in autofz after the preparation phase of the first round. It indicates why xFUZZ underperforms compared to autofz on *exiv2*.

Table 5. The Left Part Shows the Average Unique Bug Count for 6 Fuzzers Across 8 Projects in Magma, While the Right Part Displays the Average Unique Bug Count for xFUZZ and Autofz Across 9 Programs in Unifuzz.

Project	xFUZZ	AFL++		MOpt		EcoFuzz		AFLFast		AFL	
	Bug	Bug	$p/\hat{A}_{12}$	Bug	$p/\hat{A}_{12}$	Bug	$p/\hat{A}_{12}$	Bug	$p/\hat{A}_{12}$	Bug	$p/\hat{A}_{12}$
libpng	**2.2**	**2.2**	1.000/0.50	1.4	0.056/0.84	1.4	0.056/0.84	1.2	0.021/0.92	1.0	0.006/1.00
libsndfile	**7.0**	6.8	0.424/0.60	**7.0**	1.000/0.50	**7.0**	1.000/0.50	2.0	0.004/1.00	2.2	0.006/1.00
libtiff	3.0	**3.4**	0.177/0.30	**3.4**	0.177/0.30	3.0	1.000/0.50	3.0	1.000/0.50	2.4	0.067/0.80
libxml2	3.8	**4.2**	0.406/0.34	2.8	0.125/0.80	3.2	0.232/0.72	1.0	0.007/1.00	1.0	0.007/1.00
lua	**1.4**	1.0	0.177/0.70	0.8	0.121/0.76	0.8	0.121/0.76	1.0	0.177/0.70	1.0	0.177/0.70
openssl	**4.8**	3.8	0.021/0.92	3.6	0.054/0.86	4.2	0.093/0.80	4.0	0.020/0.90	1.6	0.009/1.00
php	**3.0**	2.0	0.004/1.00	2.6	0.177/0.70	1.2	0.006/1.00	2.2	0.177/0.70	2.6	0.424/0.60
poppler	3.2	2.4	0.056/0.84	**3.4**	0.600/0.40	2.0	0.006/1.00	2.0	0.006/1.00	1.8	0.007/1.00
Sum	**28.4**	25.8		25		22.8		16.4		13.6	
Best	**5**	3		3		1		0		0	

Program	xFUZZ	autofz	
	Bug	Bug	$p/\hat{A}_{12}$
cflow	**5**	3	0.024\0.90
exiv2	5.4	**12**	0.045\0.10
imginfo	**1**	0.2	0.020\0.90
infotocap	1.2	**2.2**	0.441\0.36
lame	**2.6**	1.4	0.027\0.92
mujs	0	**2.8**	0.007\0.00
nm	**0.4**	0.2	0.600\0.60
objdump	**7.6**	0.4	0.009\1.00
pdftotext	**9**	1.6	0.011\1.00
Sum	**32.2**	23.8	
Best	**6**	3	

6.2 Vulnerability Discovery Ability of xFUZZ

To evaluate the vulnerability discovery ability of xFUZZ, we compared its performance against both individual fuzzing and collaborative fuzzing approaches.

6.2.1 Comparison with Individual Fuzzing. We evaluated the performance of xFUZZ against five baseline fuzzers on Magma[17]. Magma is a benchmark suite based on real programs and real vulnerabilities, which reintroduces known vulnerabilities and statistical code into projects to analyze the ability of different fuzzers to reach and trigger vulnerabilities. We selected 15 target programs from 8 projects, including libpng (libpng_read_fuzzer), libsndfile (sndfile_fuzzer), libtiff (tiff_read_rgba_fuzzer), libxml2 (libxml2_xml_read_memory_fuzzer), lua (lua), openssl (asn1, asn1parse, bignum, server, client, x509), php (json, exif, unserialize, parser), and poppler (pdf_fuzzer). All these projects are part of the Magma suite. The experiments were conducted over 5 trials, with each trial running for 24 hours. Table 5 summarizes the average number of unique vulnerabilities discovered by each fuzzer. xFUZZ discovered the most unique vulnerabilities in 5 out of the 8 projects, totaling 28.4, surpassing the other 5 baseline fuzzers. Compared to the second-best fuzzer (25.8), xFUZZ discovered 10.07% more unique bugs.

We also measured the time to exposure (TTE) for each fuzzer (as shown in Table 6). xFUZZ was the fastest to trigger 21 of 37 unique bugs—162.5% more than AFL++, highlighting its efficiency in vulnerability detection.

6.2.2 Comparison with Collaborative Fuzzing. Due to the complexity of autofz project, it could not be integrated into Magma. Therefore, we evaluated the average number of unique vulnerabilities discovered by xFUZZ and autofz across 12 target programs in Unifuzz, with 9 of these targets successfully detecting vulnerabilities, as shown in Table 5. Notably, we used the hash values of the three layers of the function call stack in ASAN reports as identifiers for unique vulnerabilities.

xFUZZ identified the most unique vulnerabilities in 6 out of the 9 targets compared with autofz, discovering a total of 32.2 unique vulnerabilities, which is 35.29% more than autofz (23.8). Particularly, in the cases of *pdftotext* and *objdump*, xFUZZ discovered 462.5% and 1800% more unique vulnerabilities than autofz.

6.3 Exploit Time Setting for xFUZZ

xFUZZ operates in a continuous loop of exploration and exploitation phases, where it identifies optimal configurations during the exploration phase and then leverages these configurations in the exploitation phase. Determining an effective ratio of exploitation time to exploration time is essential for maximizing performance. To identify the best ratio, we evaluated different settings—0.3,

Table 6. Average Time to Exposure (TTE) for 6 Fuzzers Across 8 Projects in Magma, Over 5 Runs of 24 Hours Each

Bug ID	xFUZZ	AFL++	MOpt	EcoFuzz	AFLFast	AFL	Bug ID	xFUZZ	AFL++	MOpt	EcoFuzz	AFLFast	AFL
PNG003	**19s**	21s	31s	28s	25s	20s	SND020	**8m**	44m	12m	1h	1w	1w
XML017	**28s**	1m	39s	40s	31s	30s	XML009	**9m**	1h	59m	12h	1w	1w
PDF016	**57s**	1m	2m	2m	10m	3m	SND006	32m	1d	**9m**	2h	1w	1w
SND017	**15s**	20m	1m	1h	19m	1h	SSL001	1d	**5h**	2d	20h	1w	5d
SND005	21m	25m	**1m**	1h	1h	27m	PDF018	**16m**	4d	39m	1w	1w	1w
TIF007	2m	**59s**	4m	2m	2h	5h	SSL009	**20s**	1w	5d	1d	8h	4d
TIF012	44m	**34m**	**34m**	1h	12h	13h	PNG007	**3h**	4h	4d	4d	5d	1w
PDF010	6m	1h	52m	1h	**1m**	1d	XML001	1d	**5h**	5d	5d	1w	1w
SSL003	**15s**	3m	2m	3m	3m	2d	XML003	4d	**4h**	3d	18h	1w	1w
SSL002	**32s**	4m	3m	3m	3m	4d	XML012	**3d**	1w	1w	1w	1w	1w
PHP011	**15s**	9m	10m	3m	2d	1d	TIF002	1w	**4d**	5d	1w	1w	1w
TIF014	**1h**	3h	2h	**1h**	9h	4d	PDF008	**5d**	1w	**5d**	1w	1w	1w
PHP009	**15s**	2h	1d	1w	2d	1d	LUA003	**4d**	1w	1w	1w	1w	1w
LUA004	**4m**	5h	8h	12h	14h	15h	PNG001	1w	**5d**	1w	1w	1w	1w
SSL020	10h	1d	1d	4d	**5h**	1w	XML002	1w	**5d**	1w	1w	1w	1w
SND001	**5m**	8m	7m	15m	1w	5d	PNG006	**5d**	1w	1w	1w	1w	1w
PHP004	**15s**	1w	1d	5d	2h	1h	TIF008	1w	1w	**5d**	1w	1w	1w
SND024	3m	20m	9m	**2m**	1w	1w	PDF019	1w	1w	**5d**	1w	1w	1w
SND007	15m	20m	**9m**	46m	1w	1w	**Fastest**	**21**	8	7	2	2	0

In Magma, to determine whether a fuzzer can trigger a specific vulnerability, if a fuzzer fails to do so within 1d, the trigger time is recorded as 1w. Therefore, in the average results of 5 trials, the trigger time for some vulnerabilities may exceed 1d.

Table 7. Branch Coverage for Different xFUZZ Settings Across 12 Target Programs in Fuzzbench

Program	xFUZZ Coverage	$xFUZZ_{0.3}$ Coverage(p/$\hat{A}_{12}$)	$xFUZZ_{1}$ Coverage(p/$\hat{A}_{12}$)	$xFUZZ_{3}$ Coverage(p/$\hat{A}_{12}$)	$xFUZZ_{5}$ Coverage(p/$\hat{A}_{12}$)	$xFUZZ_{global}$ Coverage(p/$\hat{A}_{12}$)	$xFUZZ_{single}$ Coverage(p/$\hat{A}_{12}$)	$xFUZZ_{full}$ Coverage(p/$\hat{A}_{12}$)	$xFUZZ_{static}$ Coverage(p/$\hat{A}_{12}$)
bloaty	6236.3	6185.7(0.427/0.61)	6167.5(0.140/0.70)	6137.2(0.009/0.85)	6213.6(0.520/0.59)	6187(0.273/0.65)	6185.1(0.545/0.585)	6144(0.054/0.76)	5985.5(< 10^{-3}/1.0)
file	2263.7	2225.1(0.623/0.43)	2247.7(0.405/0.39)	2225.1(0.623/0.57)	2235.1(0.91/0.52)	1987.4(0.001/0.96)	2235.3(0.791/0.54)	2251.7(0.427/0.39)	2215.8(0.678/0.44)
harfbuzz	10976.2	11003.6(0.121/0.29)	10967.5(0.570/0.58)	10964.9(0.650/0.57)	10953.4(0.256/0.66)	11024(0.009/0.14)	10906.4(0.002/0.92)	10910.2(0.008/0.87)	10878.9(< 10^{-3}/0.98)
lcms	1649.6	1684.2(1.000/0.50)	1538.5(0.089/0.73)	1589(0.257/0.66)	1498.1(0.162/0.69)	1299.1(0.002/0.92)	1667.9(0.910/0.48)	1573.1(0.212/0.67)	1546.2(0.121/0.71)
libjpeg	3209.3	3209.2(0.814/0.54)	3210.1(0.355/0.38)	3210.1(0.439/0.40)	3209.5(0.375/0.38)	2543.8(< 10^{-3}/1.00)	2547.9(< 10^{-3}/1.00)	2546.4(< 10^{-3}/1.00)	2545.9(< 10^{-3}/1.00)
libpng	1987	1976(0.150/0.70)	1980.4(0.344/0.63)	1971(0.103/0.72)	1982.2(0.761/0.55)	1772.4(< 10^{-3}/1.00)	1988.1(0.733/0.45)	1985.3(0.761/0.46)	1988.4(0.761/0.46)
mbedtls	3286.8	2820.2(0.005/0.88)	2846(0.021/0.81)	2858.2(0.011/0.84)	2912.4(0.021/0.81)	2803(0.001/0.93)	2942(0.026/0.80)	2861.7(0.006/0.87)	2728.2(0.001/0.96)
mruby	7985.3	7784.8(0.026/0.80)	7935.7(0.650/0.57)	7898.5(0.273/0.65)	7799.9(0.031/0.79)	7423.4(< 10^{-3}/0.99)	8065.5(0.345/0.37)	8002.9(0.850/0.47)	7737.5(0.031/0.79)
openssl	5832.1	5831.5(0.607/0.57)	5832(1.000/0.50)	5831.2(0.337/0.63)	5831.7(0.519/0.59)	5831.7(0.658/0.59)	5832.9(0.100/0.29)	5832.8(1.000/0.50)	5833.2(0.893/0.27)
vorbis	1268.3	1265.4(0.048/0.77)	1265.7(0.108/0.72)	1266.2(0.108/0.72)	1267.2(0.361/0.63)	1266.9(0.381/0.62)	1266.7(0.234/0.66)	1266.6(0.255/0.66)	1266.3(0.093/0.73)
woff2	1131	1133.5(0.224/0.34)	1134.5(0.127/0.30)	1131.7(0.732/0.45)	1136(0.069/0.26)	1116.4(1.000/0.50)	1140.2(0.005/0.13)	1138.3(0.019/0.19)	1140.9(0.003/0.11)
zlib	460.9	458.7(0.592/0.58)	459.8(0.402/0.62)	459.3(0.700/0.56)	460.6(0.379/0.62)	449.2(0.622/0.57)	461.6(0.441/0.40)	460(0.757/0.55)	461.4(0.441/0.40)
Total	**46286.5**	45577.9	45585.4	45542.4	45499.7	43704.3	45239.6	44973	44328.2

The p-value and $\hat{A}_{12}$ measure are used to evaluate the statistical significance and effect size between $xFUZZ_{0.5}$ and other xFUZZ settings.

0.5, 1, 3, and 5—representing the proportion of exploitation time relative to exploration time, across 12 target programs from Fuzzbench. The results, presented in Table 7, show that a ratio of 0.5 yields the highest code coverage for xFUZZ on 8 out of the 12 programs. Based on these findings, xFUZZ 's default configuration sets exploitation time to 0.5 times the exploration time. Since many of the experimental differences were not statistically significant, we concluded that the impact of exploit time on xFUZZ 's performance is relatively minor, and therefore selected the exploit time setting that produced the highest coverage.

The relatively short 0.5x exploitation period might seem counterintuitive. However, our analysis suggests this configuration aligns with the tiered exploration strategy, which progressively identifies optimal plugins layer by layer. As exploration progresses, previously selected advantageous plugins are fully utilized, allowing xFUZZ to capture optimal configurations in later phases with shorter exploitation times. A shorter cycle enables xFUZZ to adapt flexibly, fine-tuning its strategy with minimal delay and ensuring consistent performance improvements across various targets.

6.4 Effectiveness of xFUZZ's Tiered Exploration Strategy

As described in Section 4.3.3, we introduce a tiered exploration strategy intended to reduce the overall exploration time. To evaluate its effectiveness, we developed a variant of xFUZZ using a

global exploration strategy, denoted as xFUZZ$_{global}$, which exhaustively explores every possible plugin composition and selects the best-performing configuration. The radio of exploitation time to exploration time is same as the original xFUZZ.

The weaker performance of xFUZZ$_{global}$ is attributed to its prolonged exploration time, which delays the identification of optimal strategy compositions. In contrast, the tiered strategy in xFUZZ enables faster, layer-by-layer discovery of effective plugins, leading to significantly better overall performance than xFUZZ$_{global}$.

6.5 Effectiveness of xFUZZ's SW-TS Algorithm

We evaluated the effectiveness of xFUZZ 's Sliding-Window Thompson Sampling (SW-TS) algorithm through ablation studies. This algorithm models strategy selection as a Non-Stationary Multi-Armed Bandit (NS-MAB) problem to dynamically adjust the fuzzing strategy composition.

To this end, we constructed two variants of xFUZZ: xFUZZ$_{single}$ and xFUZZ$_{full}$. xFUZZ$_{single}$ selects a strategy composition for exploitation based solely on the exploration results from the current round, whereas xFUZZ$_{full}$ aggregates the exploration results of all previous rounds. In contrast, xFUZZ uses a sliding window based on 5 rounds to dynamically select the optimal fuzzing strategy composition. All the variants of xFUZZ maintain the same time ratio configuration between exploitation and exploration as the original xFUZZ.

We conducted experiments on these two xFUZZ variants across 12 target programs in Fuzzbench, with the results shown in Table 7. xFUZZ with SW-TS outperformed the other variants on 6 of the 12 programs, with differences in coverage on the remaining targets generally within 1%. Furthermore, statistical analysis (p-value and $\hat{A}_{12}$) indicates that xFUZZ provides a notable advantage in balancing exploration and exploitation on certain targets (such as *libjpeg* and *harfbufzz*), as it can capture recent trends while preventing overreliance on outdated strategies.

6.6 Effectiveness of xFUZZ's Runtime-Adaptive Control

To evaluate the effectiveness of xFUZZ's runtime-adaptive control, we compared it with its static setting, xFUZZ$_{static}$. In xFUZZ$_{static}$, we use explore mode as the input scheduler, and havoc as the mutator scheduler. This configuration is the default mode of AFL++ and has been identified as the optimal setup through multiple evaluations.

The experimental results on Fuzzbench(shown in Table 7) indicate that xFUZZ outperforms xFUZZ$_{static}$ on 8 target programs, with differences on the remaining 4 targets within 1%. It demonstrates that xFUZZ's runtime-adaptive control provides a substantial performance boost.

Moreover, we analyzed the strategy composition selections of xFUZZ across 6 target programs, as illustrated in Figure 6. We observed an interesting phenomenon: in *bloaty* and *lcms*, xFUZZ predominantly selected EcoFuzz, which accounted for over 50% of the total fuzzing process.

In contrast, for *pdftotext* and *mruby*, the strategy composition selection leaned more towards fast mode and coe mode, collectively comprising approximately 60%. It indicates that the optimal strategy varies across different targets and fuzzing phases, aligning with the conclusions drawn in our motivation section.

7 Discussion

7.1 Limitations and Future Work

Despite its capabilities, xFUZZ currently includes only a limited set of plugins, primarily for input and mutation scheduling. This may limit its performance on targets requiring specialized strategies. Future work will focus on expanding the plugin set to support more advanced fuzzing techniques and improving adaptability across diverse targets.

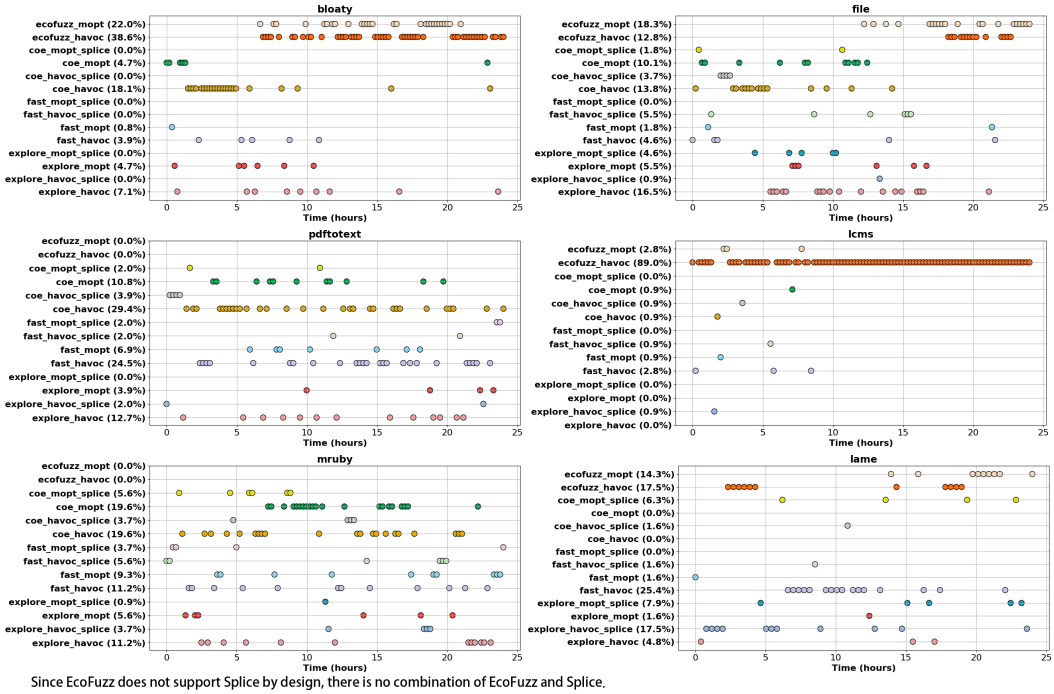


Fig. 6. The strategy composition selections of xFUZZ across 6 target programs

xFUZZ also relies solely on AFL-based instrumentation and forkserver executors. To enhance versatility, we plan to add QEMU-based executors for closed-source binaries and support distributed, cross-platform fuzzing, broadening its applicability to more complex environments.

7.2 Threats to Validity

Although our evaluation compared xFUZZ with leading fuzzers on public benchmarks, such datasets may not fully capture the complexity of real-world applications. To improve representativeness, we included additional targets such as *mruby* and *file*, aiming for a more diverse behavioral coverage.

8 Conclusion

In this paper, we presented xFUZZ, a fuzzing framework that enables fine-grained, runtime-adaptive strategy composition, addressing the limitations of existing fuzzing tools constrained by static or coarse-grained configurations. By modularizing core fuzzing components and incorporating a Thompson Sampling-based algorithm for dynamic strategy selection, xFUZZ adapts its configurations in real-time to optimize vulnerability discovery and code coverage. Experimental results show that xFUZZ outperforms state-of-the-art fuzzers, discovering 10.07% more unique vulnerabilities and achieving up to 11.48% higher code coverage.

9 Data Availability

The code, data, and analysis scripts of the paper have been made available in anonymized repositories. The repository for xFUZZ experiments, including all analysis scripts, can be accessed at

Table 8. The Left Table Shows the Coverage Results for xFUZZ$_{angora}$, xFUZZ and autofz in Unifuzz, While the Right Table Displays the Average Unique Bug Count for these three fuzzers Across 10 Programs in Unifuzz.

Program	xFUZZ$_{angora}$	xFUZZ			autofz		
	Cover	Cover	R_{cov}	$\hat{A}_{12}$	Cover	R_{cov}	$\hat{A}_{12}$
cflow	1133	1135	-0.18%	0.013/0.02	1130.4	+0.23%	0.107/0.82
exiv2	4677	3962.4	+18.03%	0.016/1.00	5572.2	-16.07%	0.016/0.00
imginfo	1937.4	1810	+7.04%	0.095/0.84	1819.8	+6.46%	0.142/0.80
infotocap	1635.2	1443.4	+13.29%	0.032/0.92	1455.6	+12.34%	0.095/0.84
jhead	418.8	201	+108.36%	0.007/1.00	193.2	+116.77%	0.012/1.00
jq	1902.2	1902.8	-0.03%	0.917/0.54	1897.4	+0.25%	0.841/0.56
lame	3885	3882.6	+0.06%	1.000/0.52	3705.6	+4.84%	0.008/1.00
mujs	3742.8	3383.8	+10.61%	0.008/1.00	3137.2	+19.30%	0.008/1.00
nm	3136.8	2998.2	+4.62%	0.008/1.00	2265.6	+38.45%	0.008/1.00
objdump	4429.6	4543.4	-2.50%	0.056/0.12	2705.8	+63.71%	0.008/1.00
pdftotext	6373	6468.8	-1.48%	0.690/0.40	6057.8	+5.20%	0.008/1.00
tcpdump	10291.8	10432	-1.34%	1.000/0.52	7882.4	+30.57%	0.008/1.00
Total	43562.6	42163.4	+3.32%	-	37823	+15.17%	-

Program	xFUZZ$_{angora}$	xFUZZ		autofz	
	Bug	Bug	$\hat{A}_{12}$	Bug	$\hat{A}_{12}$
cflow	4.2	**5**	0.020/0.10	3	0.406/0.66
exiv2	9	5.4	0.140/0.80	**12**	0.055/0.10
imginfo	0.6	**1**	0.177/0.30	0.2	0.270/0.70
infotocap	**3.8**	1.2	0.016/0.96	2.2	0.200/0.76
jhead	**11.4**	0	0.025/0.90	0	0.025/0.90
lame	**2.8**	2.6	0.600/0.60	1.4	0.015/0.96
mujs	0.4	0	0.177/0.70	**2.8**	0.029/0.08
nm	**1**	0.4	0.204/0.74	0.2	0.083/0.82
objdump	**7.6**	**7.6**	0.830/0.44	0.4	0.009/1.00
pdftotext	6	**9**	0.172/0.22	1.6	0.014/0.98
Sum	46.8	32.2		23.8	
Best	5	4		2	

https://anonymous.4open.science/r/xfuzz_experiments-FC3B. The code of xFUZZ used in the paper is available at https://anonymous.4open.science/r/xfuzz_submit-6BE0.

Acknowledgments

This work is supported in part by National Natural Science Foundation of China (U24A20337), Zhongguancun Laboratory, and Joint Research Center for System Security (JCSS), Tsinghua University (Institute for Network Sciences and Cyberspace) - Science City (Guangzhou) Digital Technology Group Co., Ltd.

A Experimental Data of xFUZZ with Angora Integrated

To verify the hypothesis in Section 6.1.2, we integrated Angora as an extension to the Mutator Scheduler in the latest version of xFUZZ. When Angora is enabled, xFUZZ extends the regular mutation logic of the Mutator Scheduler by incorporating Angora's context-sensitive bitmap, performing taint analysis on seeds, and applying additional gradient descent-based mutation strategies targeting unexplored branches.

The experimental results are shown in Table 8. We refer to the version of xFUZZ with Angora integrated as xFUZZ$_{angora}$. Compared to the prior version of xFUZZ, xFUZZ$_{angora}$ achieved an overall coverage increase of 3.32%, with more than 10% improvement in code coverage on *exiv2*, *infotocap*, *jhead*, and *mujs*. Notably, code coverage increased by 108% on *jhead*.

Additionally, xFUZZ$_{angora}$ discovered 46.8 unique vulnerabilities, representing a 45.3% increase compared to xFUZZ. On programs such as *exiv2*, *infotocap*, *jhead*, and *nm*, xFUZZ$_{angora}$ significantly found more unique vulnerabilities, especially on *jhead*, where its substantial code coverage improvement led to the discovery of 11.4 new unique vulnerabilities. We believe that the ability of xFUZZ$_{angora}$ to discover more unique vulnerabilities can be attributed to the significant role played by Angora's context-sensitive bitmap, which allows it to capture all bugs with different function stacks triggered at the same location. Compared to autofz, xFUZZ$_{angora}$ discovered nearly twice as many unique vulnerabilities and outperformed autofz across 8 programs.

Notably, while xFUZZ$_{angora}$ outperformed xFUZZ on *exiv2* with an 18.03% increase in coverage and 66.67% more bugs found, it still lags behind autofz by 16.07% in coverage and 25% in bug discovery. This gap may stem from specialized strategies in autofz not yet available in xFUZZ. Future work will focus on integrating such strategies as plugins to enhance xFUZZ 's performance across diverse targets.

References

[1] 2016. Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com/.

[2] P. Amini. 2017. Sulley fuzzing framework. https://github.com/OpenRCE/sulley Accessed: 30 Oct 2024.

[3] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. *Proceedings 2019 Network and Distributed System Security Symposium* (2019). https://api.semanticscholar.org/CorpusID:85546717

[4] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. 2002. Finite-time Analysis of the Multiarmed Bandit Problem. *Mach. Learn.* 47, 2–3 (may 2002), 235–256. https://doi.org/10.1023/A:1013689704352

[5] Tim Blazytko, Matt Bishop, Cornelius Aschermann, Justin Cappos, Moritz Schlögel, Nadia Korshun, Ali Abbasi, Marco Schweighauser, Sebastian Schinzel, Sergej Schumilo, et al. 2019. {GRIMOIRE}: Synthesizing structure while fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*. 1985–2002.

[6] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. 2020. Boosting fuzzer efficiency: an information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 678–689. https://doi.org/10.1145/3368089.3409748

[7] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428

[8] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*. 711–725. https://doi.org/10.1109/SP.2018.00046

[9] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. EnFuzz: ensemble fuzzing with seed synchronization among diverse fuzzers. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 1967–1983.

[10] Richard Combes and Alexandre Proutiere. 2014. Unimodal Bandits: Regret Lower Bounds and Optimal Algorithms. In *Proceedings of the 31st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 32)*, Eric P. Xing and Tony Jebara (Eds.). PMLR, Bejing, China, 521–529. https://proceedings.mlr.press/v32/combes14.html

[11] Joeri De Ruiter and Erik Poll. 2015. Protocol state fuzzing of {TLS} implementations. In *24th USENIX Security Symposium (USENIX Security 15)*. 193–206.

[12] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

[13] Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and Davide Balzarotti. 2022. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *Proceedings of the 29th ACM conference on Computer and communications security (CCS)* (Los Angeles, U.S.A.) *(CCS '22)*. ACM.

[14] Yu-Fu Fu, Jaehyuk Lee, and Taesoo Kim. 2023. autofz: Automated Fuzzer Composition at Runtime. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 1901–1918. https://www.usenix.org/conference/usenixsecurity23/presentation/fu-yu-fu

[15] Patrice Godefroid. 2007. Random testing for security: blackbox vs. whitebox fuzzing. In *Proceedings of the 2nd International Workshop on Random Testing: Co-Located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)* (Atlanta, Georgia) *(RT '07)*. Association for Computing Machinery, New York, NY, USA, 1. https://doi.org/10.1145/1292414.1292416

[16] Emre Güler, Philipp Görz, Elia Geretto, Andrea Jemmett, Sebastian Österlund, Herbert Bos, Cristiano Giuffrida, and Thorsten Holz. 2020. Cupid: Automatic Fuzzer Selection for Collaborative Fuzzing. In *Proceedings of the 36th Annual Computer Security Applications Conference* (Austin, USA) *(ACSAC '20)*. Association for Computing Machinery, New York, NY, USA, 360–372. https://doi.org/10.1145/3427228.3427266

[17] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 3, Article 49 (nov 2020), 29 pages. https://doi.org/10.1145/3428334

[18] A. Helin. 2021. Radamsa. https://gitlab.com/akihe/radamsa.

[19] Patrick Jauernig, Domagoj Jakobovic, Stjepan Picek, Emmanuel Stapf, and Ahmad-Reza Sadeghi. 2023. DARWIN: Survival of the Fittest Fuzzing Mutators. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/darwin-survival-of-the-fittest-fuzzing-mutators/

[20] Myungho Lee, Sooyoung Cha, and Hakjoo Oh. 2023. Learning Seed-Adaptive Mutation Strategies for Greybox Fuzzing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 384–396. https://doi.org/10.1109/ICSE48619.2023.00043

[21] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 475–485. https://doi.org/10.1145/3238147.3238176

[22] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity* 1, 1 (2018), 6. https://doi.org/10.1186/s42400-018-0002-y

[23] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. 2021. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. In *Proceedings of the 30th USENIX Security Symposium*.

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

[25] Jonathan Metzman, László Szekeres, Laurent Maurice Romain Simon, Read Trevelin Sprabery, and Abhishek Arya. 2021. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1393–1403. https://doi.org/10.1145/3468264.3473932

[26] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (dec 1990), 32–44. https://doi.org/10.1145/96267.96279

[27] Roberto Natella. 2022. Stateafl: Greybox fuzzing for stateful network servers. *Empirical Software Engineering* 27, 7 (2022), 191.

[28] Sebastian Österlund, Elia Geretto, Andrea Jemmett, Emre Güler, Philipp Görz, Thorsten Holz, Cristiano Giuffrida, and Herbert Bos. 2021. CollabFuzz: A Framework for Collaborative Fuzzing. In *Proceedings of the 14th European Workshop on Systems Security* (Online, United Kingdom) *(EuroSec '21)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3447852.3458720

[29] PeachTech. 2017. Peach. https://www.peach.tech/ Accessed: 30 Oct 2024.

[30] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 460–465.

[31] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Network and Distributed System Security Symposium*. https://api.semanticscholar.org/CorpusID:2354736

[32] Dongdong She, Abhishek Shah, and Suman Jana. 2022. Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2194–2211. https://doi.org/10.1109/SP46214.2022.9833761

[33] Ilja Van Sprundel. 2005. Fuzzing: Breaking software in an automated fashion. (2005).

[34] Prashast Srivastava and Mathias Payer. 2021. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th acm sigsoft international symposium on software testing and analysis*. 244–256.

[35] SSLab@Gatech. 2023. Autofz Project. https://github.com/sslab-gatech/autofz Accessed: 23 Feb 2025.

[36] William R Thompson. 1933. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika* 25, 3/4 (1933), 285–294.

[37] Francesco Trovò, Stefano Paladino, Marcello Restelli, and Nicola Gatti. 2020. Sliding-Window Thompson Sampling for Non-Stationary Settings. *Journal of Artificial Intelligence Research* 68 (05 2020), 311–364. https://doi.org/10.1613/jair.1.11407

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

[39] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. USENIX Association, Chaoyang District, Beijing, 1–15. https://www.usenix.org/conference/raid2019/presentation/wang

[40] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. *Proceedings 2020 Network and Distributed System Security Symposium* (2020). https://api.semanticscholar.org/CorpusID:211268394

[41] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. 2022. One fuzzing strategy to rule them all. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1634–1645. https://doi.org/10.1145/3510003.3510174

[42] Tai Yue, Yong Tang, Bo Yu, Pengfei Wang, and Enze Wang. 2019. LearnAFL: Greybox Fuzzing With Knowledge Enhancement. *IEEE Access* 7 (2019), 117029–117043. https://doi.org/10.1109/ACCESS.2019.2936235

[43] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2307–2324. https://www.usenix.org/conference/usenixsecurity20/presentation/yue

[44] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 745–761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[45] Michał Zalewski. 2016. American Fuzzy Lop - Whitepaper. https://lcamtuf.coredump.cx/afl/technical_details.txt. [Online; accessed 8 August 2024].

[46] Kunpeng Zhang, Xiaogang Zhu, Xi Xiao, Minhui Xue, Chao Zhang, and Sheng Wen. 2024. ShapFuzz: Efficient Fuzzing via Shapley-Guided Byte Selection. In *Proceedings 2024 Network and Distributed System Security Symposium (NDSS 2024)*. Internet Society. https://doi.org/10.14722/ndss.2024.23134

Received 2025-02-27; accepted 2025-03-31