SimiFuzz: Seed–Worker Scheduling for Parallel Fuzzing via Contextual Bandits

YIJIA GUO, Zhejiang Normal University, China
ZHIGUO DING, Zhejiang Normal University, China
HONG LIANG, Zhejiang University, China
MING ZHONG, Zhejiang Normal University, China
DANDAN ZHAO, Zhejiang Normal University, China
XUHONG ZHANG, Zhejiang University, China
BO ZHANG, China Electric Power Research Institute, China
SHOULING JI, Zhejiang University, China
HAO PENG*, Zhejiang Normal University, China

Parallel fuzzing is now a standard way to scale vulnerability discovery, yet its efficiency is still limited by ineffective task allocation among workers. Existing approaches mainly aim to reduce conflicts; however, none considers the interaction between seeds and workers: the same seed can yield very different gains on different workers due to their divergent exploration states. As a result, parallel fuzzing can drift toward over-isolation that wastes shared states, or excessive overlap that duplicates effort.

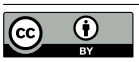
To solve this problem, we present SimiFuzz, a context-aware scheduling framework that learns to assign *seed–worker* pairs online. SimiFuzz encodes each assignment with a compact context vector that jointly models seed characteristics, worker state, and seed–worker interaction. On top of this representation, SimiFuzz employs a LinUCB-based contextual bandit to score candidate pairs, balancing individual worker efficiency against group-level redundancy to maximize collective progress. To handle non-stationary fuzzing dynamics, SimiFuzz adopts a time-slice feedback mechanism that aggregates coverage gains within fixed intervals, combining globally new edges with cross-learning progress to form stable reward signals. We implement SimiFuzz on top of AFL++ and evaluate it on eight real-world targets. In 24-hour campaigns with 10 parallel instances, SimiFuzz improves average edge coverage by 11.76 % over FlexFuzz, the strongest baseline in coverage and unique vulnerability (VUL) count, achieves the highest final coverage on all evaluated targets, and uncovers 16 more unique vulnerabilities and 11 more CVEs than FlexFuzz.

CCS Concepts: • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: Parallel Fuzzing, Seed Scheduling, Contextual Multi-Armed Bandit, Task Allocation, Coverage-Guided Fuzzing

*Corresponding author

Authors' Contact Information: Yijia Guo, de3mond@zjnu.edu.cn, Zhejiang Normal University, Jinhua, China; Zhiguo Ding, dzg@zjnu.edu.cn, Zhejiang Normal University, Jinhua, China; Hong Liang, hongliang@zju.edu.cn, Zhejiang University, Hangzhou, China; Ming Zhong, zhongming@zjnu.edu.cn, Zhejiang Normal University, Jinhua, China; Dandan Zhao, ddzhao@zjnu.edu.cn, Zhejiang Normal University, Jinhua, China; Xuhong Zhang, zhangxuhong@zju.edu.cn, Zhejiang University, Ningbo, China; Bo Zhang, zhangbo6@epri.sgcc.com.cn, China Electric Power Research Institute, Beijing, China; Shouling Ji, sji@zju.edu.cn, Zhejiang University, Hangzhou, China; Hao Peng, hpeng@zjnu.edu.cn, Zhejiang Normal University, Jinhua, China.



ACM 2994-970X/2026/10-ARTISSTA016
https://doi.org/10.1145/3832107

ACM Reference Format:
Yijia Guo, Zhiguo Ding, Hong Liang, Ming Zhong, Dandan Zhao, Xuhong Zhang, Bo Zhang, Shouling Ji, and Hao Peng. 2026. SimiFuzz: Seed–Worker Scheduling for Parallel Fuzzing via Contextual Bandits. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA016 (October 2026), 22 pages. https://doi.org/10.1145/3832107

1 Introduction

Fuzzing has emerged as a highly effective technique for software vulnerability discovery and is now widely adopted in security testing [11, 40]. Its core principle involves continuously generating a large volume of test cases to explore the program state space, thereby triggering potential vulnerabilities or abnormal behaviors [17, 26]. However, as modern software systems continue to grow in scale and complexity, the program state space expands dramatically, making single-instance fuzzing insufficient for efficient testing. Parallel fuzzing deploys multiple instances to explore different program regions simultaneously. It has become a mainstream solution to this challenge and is widely adopted in industrial practice [7, 10, 31].

In the early practice of parallel fuzzing, commonly used AFL-style fuzzers [11, 40] launch multiple identical fuzzing instances and use a synchronization directory to share test cases. However, this standard parallelization model is plagued by exploration overlap, often causing different instances to repeatedly explore the same program regions, resulting in a mismatch between testing efficiency and computational resources. To address this issue, researchers have proposed the concept of task partitioning [21], which divides the program's exploration space into subtasks and assigns them to different fuzzing instances, thereby reducing exploration overlap.

Categorized by the timing of task assignment, existing task partitioning methods are broadly classified as static and dynamic approaches, as shown in Table 1. Static partitioning methods perform a one-time partition before testing begins according to predefined rules. For example, PAFL [21] partitions based on coverage bitmap blocks. These methods are straightforward to implement. However, their partitioning is fixed once decided, so they cannot adapt to runtime coverage changes. As a result, the assigned tasks may no longer match the actual exploration progress as fuzzing evolves. To address this limitation, dynamic partitioning methods have emerged, which periodically adjust partitioning strategies at runtime to adapt to evolving exploration states. For instance, FlexFuzz [20] employs boundary-sensitive partitioning, and DynamiQ [38] dynamically repartitions based on coverage feedback.

Although these methods have made progress in reducing exploration overlap, they primarily focus on partitioning the exploration space through seed metrics or spatial constraints. However, effective scheduling in parallel fuzzing demands a multi-dimensional perspective rather than treating each instance independently. Seed properties remain fundamental, as they determine the intrinsic exploration potential of each test case. Worker states become critical, since each instance accumulates different coverage and develops distinct exploration directions over time. Most importantly, seed–worker interactions emerge as a unique concern: the same seed may yield substantial coverage gains on one worker but cause redundant exploration on another that has already covered similar regions. Existing methods largely overlook these multi-dimensional dependencies, making it difficult to assess the expected utility of scheduling decisions or perceive real-time overlap among workers. We argue that efficient parallel fuzzing requires balancing individual efficiency and collective diversity. Specifically, the scheduler should maximize the coverage gain of each seed–worker assignment while simultaneously minimizing exploration overlap among workers. Addressing this goal presents the following challenges:

Challenge 1: How to evaluate scheduling utility and perceive exploration overlap? Accurately assessing scheduling utility requires understanding the complex interplay among seed properties, worker states, and their pairwise interactions, all of which are mutually dependent and

evolve continuously during testing. Perceiving overlap is similarly difficult, as conflict information is implicitly distributed across workers' coverage states and requires real-time aggregation. Existing methods typically rely on simple seed-level heuristics or static spatial partitioning, failing to capture such dynamic, multi-dimensional runtime information.

Challenge 2: How to achieve an adaptive balance between individual efficiency and collective diversity? The optimal trade-off strategy is not static but varies dynamically with target program characteristics and testing phases. Different programs exhibit different structural complexities and execution characteristics, and even the same program requires different emphases during early versus late testing stages. Existing methods typically rely on predefined fixed rules or manually configured weights, unable to adapt to runtime states.

To address these challenges, we propose SimiFuzz, a context-aware learning-based scheduling framework for parallel fuzzing. For **Challenge 1**, we design a multi-dimensional context feature vector that models scheduling decisions from three perspectives: seed properties, worker states, and seed–worker interactions. Seed property features characterize the intrinsic exploration potential of seeds, serving as the basis for utility evaluation. Worker state features reflect the current coverage capability and growth trend of each fuzzing instance. Interaction features capture the matching relationship between seeds and specific workers, while also encoding coverage overlap among workers to enable real-time perception of exploration redundancy. Together, these features provide a unified representation for accurate assessment of scheduling utility and exploration overlap. For **Challenge 2**, we employ a contextual multi-armed bandit algorithm as the scheduling mechanism, which automatically learns the relative importance of each feature dimension from runtime feedback, thereby adapting to the characteristics of different target programs and the varying phases of the testing process. Furthermore, we design a reward function aligned with the balancing objective, jointly considering coverage gains and exploration diversity to guide the scheduling strategy toward continuously seeking an optimal balance between individual efficiency and collective diversity.

We implement SimiFuzz on top of AFL++ and conduct a rigorous evaluation on 8 real-world programs against 4 state-of-the-art parallel fuzzing tools. The results demonstrate that SimiFuzz improves average edge coverage by 11.76 % over FlexFuzz, the strongest baseline in coverage and VUL count, and discovers 16 more unique vulnerabilities and 11 more CVEs than FlexFuzz.

In summary, this paper makes the following contributions:

- **Interaction-aware Scheduling Framework:** We formalize task scheduling in parallel fuzzing as a contextual multi-armed bandit problem, jointly modeling seed properties, worker states, and their interactions to enable adaptive seed–worker matching through online learning.
- **Implementation of SimiFuzz:** We implement a prototype of SimiFuzz on top of AFL++. The design exhibits good generality and can be extended to other fuzzing tools. We will open source our implementation to facilitate future research.
- **Comprehensive Evaluation:** We conduct an extensive evaluation on eight real-world programs. Compared with FlexFuzz, the strongest baseline in coverage and VUL count, SimiFuzz achieves an 11.76 % gain in average edge coverage and finds 16 additional unique vulnerabilities and 11 additional CVEs.

2 Background and Motivation

2.1 Background

2.1.1 Parallel Fuzzing. Parallel fuzzing addresses the efficiency bottleneck of single-instance fuzzing by deploying multiple instances simultaneously. Table 1 summarizes the characteristics of representative parallel fuzzing approaches along these dimensions. In the native parallel mode

Table 1. Comparison of parallel fuzzing systems.

Parallel Fuzzer	Task	Static	Dynamic	Structure	Context	Adaptive
AFL++ [11]	–					
PAFL [21]	Bitmap	✓				
AFL-EDGE [36]	Seed		✓			
P-Fuzz [33]	Seed		✓			
UltraFuzz [42]	Seed		✓			
μFUZZ [8]	Seed		✓			
glibFuzzer [13]	Corpus		✓			
AFLTeam [29]	Function		✓	✓		
FlexFuzz [20]	Boundary Basic Block		✓	✓		
DynamiQ [38]	Function		✓	✓		
KRAKEN [41]	Seed		✓			✓
SimiFuzz	Seed–Worker		✓		✓	✓

"Task": Task granularity (Bitmap/Seed/Corpus/Function/Boundary Basic Block); "Static"/"Dynamic": Static or dynamic task allocation; "Structure": Utilizes program structure (Control Flow Graph/Call Graph); "Context": Context-aware modeling (seed–worker interaction); "Adaptive": Adaptive learning-based scheduling.

of AFL [40] and AFL++ [11], instances operate independently and periodically synchronize seeds through a shared directory. While straightforward, this approach results in task conflict, causing instances to repeatedly explore overlapping regions [20, 21].

To mitigate task conflict, researchers have proposed various allocation strategies. Static approaches perform one-time partitioning: PAFL [21] divides the coverage bitmap into blocks assigned to distinct instances. These methods are simple but cannot adapt to runtime coverage changes.

Dynamic approaches adjust allocation at runtime. Seed-level methods such as AFL-EDGE [36], P-Fuzz [33], UltraFuzz [42], μFUZZ [8] and glibFuzzer [13] employ centralized scheduling to dispatch seeds globally, but primarily consider seed properties with limited awareness of worker states. Structure-aware methods leverage program information: AFLTeam [29] partitions call graphs into subgraphs for each worker, FlexFuzz [20] performs boundary-sensitive allocation, and DynamiQ [38] improves partitioning with entropy-weighted scoring. However, these methods rely on static analysis and incur overhead for partition computation. Adaptive methods like KRAKEN [41] dynamically adjust global strategy parameters using optimization techniques, but target macro-level configurations rather than individual scheduling decisions. Additionally, ensemble fuzzing frameworks such as EnFuzz [7] and CollabFuzz [28] combine diverse fuzzers with seed synchronization to improve coverage, though they focus on fuzzer collaboration rather than fine-grained task allocation.

Despite these advances, existing methods share a common limitation: they primarily focus on how to partition the exploration space or how to distribute seeds, while lacking direct measurement of runtime collaborative states. Specifically, it remains difficult to accurately assess the expected utility of a scheduling decision, which depends not only on the intrinsic properties of a seed but also on the current state of the target worker and the matching relationship between them. Furthermore, perceiving the degree of exploration overlap among workers in real time remains challenging, as conflict information is implicitly distributed across the coverage states of different workers and evolves continuously as testing progresses.

2.1.2 Multi-Armed Bandit. The multi-armed bandit (MAB) problem is a classical framework for sequential decision-making under uncertainty [15, 30]. In this setting, an agent repeatedly selects from a set of arms, each associated with an unknown reward distribution, with the goal of maximizing cumulative reward over time. The central challenge lies in balancing exploration of less-tested arms with exploitation of arms that have historically yielded high rewards. The Upper Confidence Bound (UCB) algorithm addresses this trade-off by selecting the arm with the highest upper confidence bound, which combines the estimated expected reward with an exploration bonus that decreases as an arm is sampled more frequently [2, 3]. While effective, traditional MAB assumes that each arm's reward distribution is stationary and independent of the decision context. Contextual MAB extends this framework by incorporating side information: at each round, the agent observes a context feature vector before making a decision, and the expected reward of each arm depends on the current context [9]. The LinUCB algorithm, a representative contextual bandit method, assumes that the expected reward is a linear function of the context features and adaptively learns the feature weights through online updates [1, 18]. This approach has been successfully applied in domains such as personalized recommendation and online advertising [6], where decisions must account for contextual information.

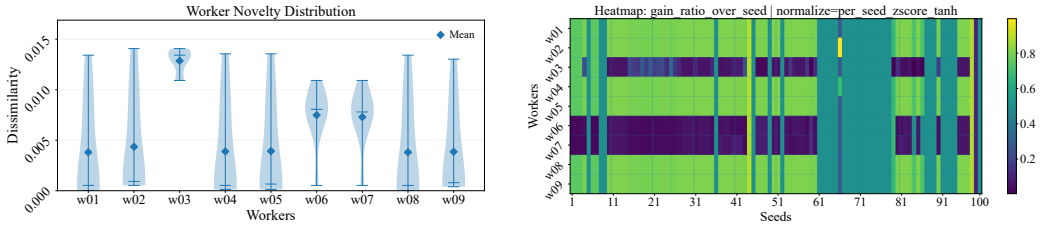
In recent years, MAB and its variants have been increasingly adopted in fuzzing to optimize various scheduling decisions. For seed scheduling, EcoFuzz [39] models seed selection as an adversarial MAB problem, dynamically adjusting selection probabilities based on historical coverage gains. AFL-HIER [35] employs multi-level coverage metrics with bandit-based scheduling to prioritize seeds hierarchically. T-Scheduler [23] applies Thompson Sampling to achieve adaptive seed prioritization without hyperparameter tuning. SyzVegas [34] extends MAB to kernel fuzzing, adapting both task and seed selection in Syzkaller. For energy allocation, SLIME [25] uses the UCB-V algorithm to select property-aware seed queues adaptively. For mutation scheduling, HavocMAB [37] models the Havoc mutation strategy as a two-layer MAB problem to dynamically adjust mutation operators. These works demonstrate the effectiveness of bandit-based approaches in single-instance fuzzing, particularly in balancing exploration with exploitation.

However, existing applications of MAB in fuzzing primarily target single-instance scenarios, where scheduling decisions are based solely on seed properties such as coverage contribution and execution frequency. In parallel fuzzing, effective scheduling requires jointly considering seed properties, worker states, and the matching relationship between them, while also balancing individual scheduling utility with global exploration diversity. To date, no prior work has applied contextual MAB to address this multi-dimensional scheduling problem in parallel fuzzing.

2.2 Motivation

Parallel fuzzing commonly improves efficiency through static or dynamic task partitioning, but existing approaches fail to effectively balance *individual efficiency* (the coverage gain potential when a worker executes a seed) against *group-level redundancy* (the degree of exploration overlap among workers), leading to suboptimal resource utilization.

For static partitioning, tasks are fixed before execution and cannot adapt to runtime coverage evolution, causing mismatch between assignments and actual exploration states. This often results in resource misallocation and redundant effort, some workers may exhaust their assigned regions while others remain stuck on hard-to-penetrate paths. For dynamic partitioning, although periodic adjustments can alleviate some overlap, most methods still rely on fixed heuristics that cannot directly estimate the expected utility of each scheduling decision or explicitly characterize the collaboration state among workers. Consequently, they cannot adaptively balance collaboration and de-duplication across different target programs and fuzzing phases.



(a) Worker novelty distribution.

(b) Seed–worker contribution heatmap.

Fig. 1. **Seed contribution and worker diversity. (a)** Worker novelty distribution: for each worker i, we plot the distribution of dissimilarity $1 - J(W_i, W_j)$ over all $j \neq i$, where J is Jaccard similarity on edge sets. Violin width indicates density, and the embedded box summarizes the median and interquartile range; larger values indicate lower overlap with other workers. **(b)** Seed–worker contribution heatmap: rows are workers, columns are seeds (1–100). Cell color indicates $gain(s, w)$ normalized per seed (z-score followed by tanh) to emphasize within-seed differences across workers; warmer colors indicate higher relative marginal contribution. Each column therefore shows the relative gain distribution of one seed across workers.

To illustrate this problem concretely, we design a compact case study to expose how parallel workers diverge during fuzzing and why the affinity between seeds and workers affects subsequent exploration. Such a case study requires a target that is complex enough for workers to develop distinct coverage states, but still manageable enough for the resulting behavior to be inspected and explained clearly. We therefore select `pdftotext` with AFL++: as a mature document parser for structured PDF inputs, it provides diverse parsing paths and a rich exploration space, making both worker divergence and seed–worker affinity observable in a compact setting. We run 10 cores (one master instance for synchronization plus nine parallel workers) for 4 hours and repeat the experiment for 5 trials, examining the phenomenon from two perspectives: worker exploration state and seed–worker contribution. For worker exploration state, we define the *dissimilarity* between two workers as $dissim(W_i, W_j) = 1 - J(W_i, W_j)$, where J denotes Jaccard similarity over edge sets. This metric quantifies how different two workers' coverage sets are; larger values indicate lower overlap. As shown in Figure 1a, we plot each worker's dissimilarity distribution with respect to all other workers. The distributions vary noticeably across workers, indicating that after a period of parallel execution, workers naturally diverge into distinct exploration states. For seed–worker contribution, we sample 100 candidate seeds from the global seed pool that have not yet been executed by any worker. For each seed, we execute it on every worker with a fixed 300 ms time budget per seed–worker pair, and record the slice-level edge summary. We define the contribution of seed s to worker w as $gain(s, w) = |S \setminus W|/|S|$, where S is the set of edges triggered by the seed within the time slice, and W is the set of edges already covered by the worker. This metric measures what fraction of the seed's triggered edges are novel to that worker. As shown in Figure 1b, the heatmap reveals that the same seed yields systematically different contributions across workers, with some seeds bringing relatively higher gains on certain workers and lower gains on others, and different seeds exhibit clearly separated gain profiles.

These results confirm the existence of observable *Seed–Worker Affinity* in parallel fuzzing: a seed's value is not an absolute property but depends on the worker's current coverage state and its overlap with other workers. We therefore adopt this affinity as the basis for context-aware scheduling, treating each decision as selecting a seed–worker pair. The scheduler leverages worker state and seed properties to predict potential gain, while incorporating inter-worker overlap information to suppress redundant exploration. This enables dynamic adaptation between individual efficiency

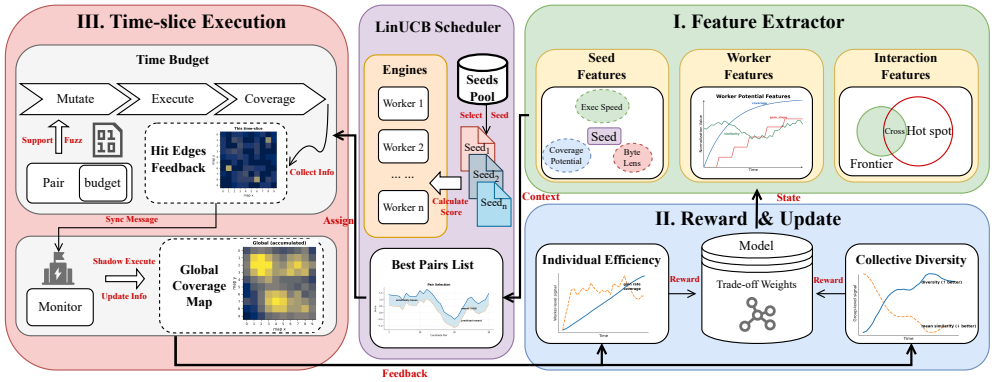


Fig. 2. **Overview of SimiFuzz.** SimiFuzz consists of three main components: Feature Extractor extracts context features from three perspectives: seed properties, worker states, and seed–worker interactions; Reward and Update module computes dual-objective rewards balancing individual efficiency and collective diversity, then updates the LinUCB model; Time-slice Execution runs parallel fuzzing instances with fixed time budgets and maintains a global coverage map. The central LinUCB Scheduler leverages extracted features to make seed–worker assignment decisions, selecting the best pairs from the seed pool. The system forms a closed-loop: execution feedback flows back to update both the global coverage map and the bandit model, enabling online adaptive scheduling.

and collective diversity throughout the fuzzing process. We design the SimiFuzz framework to realize this goal, which we detail in the next section.

3 Design

This section presents the design of SimiFuzz. We first provide an overview of the system architecture (§ 3.1), then detail the three core components: multiple-perspective feature extraction (§3.2), contextual bandit-based scheduling (§ 3.3), and time-slice feedback mechanism (§ 3.4).

3.1 Overview

Figure 2 illustrates the overall architecture of SimiFuzz. The system operates as a closed-loop online learning framework that continuously optimizes seed–worker assignments based on runtime feedback. SimiFuzz comprises three main components working in coordination with a central LinUCB scheduler.

The Feature Extractor constructs context vectors from three complementary perspectives: seed properties, worker states, and seed–worker interactions. The LinUCB Scheduler treats seed–worker assignment as a contextual bandit problem, computing Upper Confidence Bound scores for candidate pairs to balance exploitation and exploration, then selecting the best matches from the seed pool. The Reward and Update module evaluates assignment quality through a dual-objective function combining individual efficiency and collective diversity, using the resulting reward signal to update model parameters. The Time-slice Execution component runs parallel fuzzing workers within fixed time budgets, while a monitor process maintains a global coverage map that aggregates discoveries across all workers.

These components interact through a continuous feedback loop: the feature extractor gathers state information from the seed pool, workers, and global coverage map; the scheduler assigns seeds to workers based on UCB scores; workers execute their assignments and report coverage feedback;

the reward module updates the bandit model accordingly; and newly discovered seeds are added to the pool for subsequent rounds. This online learning cycle enables SimiFuzz to progressively learn scheduling policies tailored to each target program. The functionality of each module is described in the following sections.

3.2 Multiple-Perspective Feature Extraction

To make scheduling decisions that are both adaptive and lightweight, SimiFuzz represents each assignment action as a *seed–worker* pair (s, w) with a compact context vector $x(s, w)$. The key requirement is to capture intrinsic seed properties, the evolving state of each worker, and the interaction between a seed and a worker, while keeping feature computation incremental and stable for online learning. We structure the context as

$$x(s, w) = \left[x^{\text{seed}}(s),\ x^{\text{inst}}(w),\ x^{\text{int}}(s, w)\right] \in \mathbb{R}^8, \tag{1}$$

where the three blocks encode seed features (3 dims), worker features (3 dims), and interaction features (2 dims), respectively.

Seed features. We extract three seed-level features to reflect cost and potential. Two of them measure basic cost properties: seed length and execution speed. Both signals are heavy-tailed across targets and across stages of a fuzzing run, so we normalize them with a robust Z-score on log-transformed values. Specifically, for a raw measurement v of seed s, we maintain online estimates of the mean μ and standard deviation σ of $\log v$ and compute

$$z(v) = \frac{\log v - \mu}{\max(\sigma, \sigma_{\min})}, \qquad \text{Norm}(v) = \frac{\text{clip}(z(v), -z_{\max}, z_{\max}) + z_{\max}}{2z_{\max}}. \tag{2}$$

We then set $\text{len\_norm}(s) = \text{Norm}(\ell(s))$ and $\text{speed\_norm}(s) = \text{Norm}(1/\text{exec\_us}(s))$, where $\ell(s)$ is the seed length in bytes and $\text{exec\_us}(s)$ is the measured execution time. The variance floor $\sigma_{\min}$ prevents unstable normalization when the early corpus is homogeneous, and the clipping threshold $z_{\max}$ limits the influence of extreme outliers. As a result, both features remain in $[0, 1]$ and preserve ordering in the typical range without being dominated by rare extremes.

The third seed feature captures *birth-time novelty*, which describes how much new behavior the seed contributed when it was first discovered. First discovered refers to the moment when seed s is first accepted into the global seed pool after being generated by any worker, rather than later re-selection, synchronization, or re-execution by other workers. When a seed s is ingested, the worker reports its hit-edge set E_s as a snapshot of exercised edges. At this moment, the global covered-edge set E_G^{birth} denotes the union of all edges observed by all workers so far, immediately before s is inserted into the global seed pool. We compute the *frontier ratio* as the fraction of edges in E_s that were not previously covered globally, and apply a square-root smoothing:

$$\text{novel\_frac}(s) = \sqrt{\frac{|E_s \setminus E_G^{\text{birth}}|}{\max(1, |E_s|)}}. \tag{3}$$

Intuitively, novel_frac favors seeds that initially triggered uncovered regions, while the smoothing prevents a small number of extremely novel seeds from overwhelming the model in early stages. This quantity is computed once at the seed's admission time and then kept fixed as a static seed feature, even if the same seed is later selected or replayed by other workers.

Worker features. To characterize each worker w, we extract three worker-level signals: coverage, recent progress, and redundancy relative to other workers. Coverage is computed as the number of edges already covered by w divided by the map size. This feature differentiates workers that are still exploring broadly from those that have already gone deep into a local region. This difference often changes the marginal utility of assigning new seeds.

Recent progress is captured by an exponentially-smoothed gain signal. For each time slice, we compute a gain value $g_t(w)$ that reflects newly covered edges contributed by w in slice t. We maintain an exponential moving average

$$m_t(w) = \beta\, m_{t-1}(w) + (1-\beta)\, g_t(w), \qquad \text{gain\_slope}(w) = \frac{\tanh(m_t(w)) + 1}{2}. \tag{4}$$

The decay β controls the stability–responsiveness trade-off, and the tanh mapping keeps the feature bounded while preserving monotonicity. This feature allows the scheduler to react to workers whose coverage growth is stalling, while avoiding noisy oscillations caused by short-term fluctuations.

Redundancy is measured by a similarity score based on *hot edges*. Hot edges are defined as the set of *new edges discovered by a worker within a recent time window*, stored as a bitmap and periodically reset. We adopt a fixed-length sliding window for hot edges as the default configuration, with its length empirically selected to reflect recent exploration behavior without making worker-similarity and overlap estimates overly sensitive to short-term fluctuations. This is crucial because using cumulative coverage directly can severely distort similarity: long-running workers accumulate large covered sets, which can make Jaccard similarity artificially small even when workers recently explore the same region. Let H_w be worker w's hot-edge set and let $H_{\neg w}$ be the union of hot edges from all other workers. We define

$$\text{sim\_to\_group}(w) = \frac{|H_w \cap H_{\neg w}|}{\max(1, |H_w \cup H_{\neg w}|)}. \tag{5}$$

A higher value indicates that w is recently exploring overlapping regions with the group, whereas a lower value suggests more diverse exploration.

Interaction features. Finally, we include two interaction-level features to model that the same seed can have different values for different workers. The birth-time novelty above captures global newness at discovery. Complementary to that, we compute a decision-time *cross-learning* feature that estimates how much of a seed's behavior is already known globally but still missing for the target worker. Let E_G^{now} be the current global covered-edge set and E_w be the set covered by worker w. We define

$$\text{cross}(s, w) = \frac{|(E_s \cap E_G^{\text{now}}) \setminus E_w|}{\max(1, |E_s|)}. \tag{6}$$

Unlike birth-time novelty, which uses the snapshot E_G^{birth} at seed admission time, the cross-learning feature uses the current global state E_G^{now} at the moment when the scheduler evaluates a seed–worker pair. This feature is high when a seed exercises edges that many workers have already reached but worker w has not, which helps the scheduler allocate seeds that can transfer knowledge across workers and reduce duplicated effort.

All features are derived from runtime metadata and bitmap-maintained sets updated incrementally. The global and per-worker covered sets are tracked as exact bitmaps. Hot-edge sets are updated whenever new edges are discovered and are periodically reset by a fixed window so that similarity reflects *recent* exploration. This design avoids expensive corpus scans and keeps per-decision feature extraction lightweight, enabling SimiFuzz to score candidate pairs online.

3.3 Contextual Bandit-Based Scheduling

Figure 3 illustrates the contextual bandit-based scheduling process. Given the context representation $x(s, w)$, SimiFuzz formulates online seed assignment as a contextual bandit problem: at each decision point, the scheduler must choose a *seed–worker* pair (s, w) that is expected to yield high exploration benefit under limited feedback. The main goal is adaptivity to non-stationary fuzzing dynamics and lightweight decision-making that can be invoked frequently without dominating execution time.

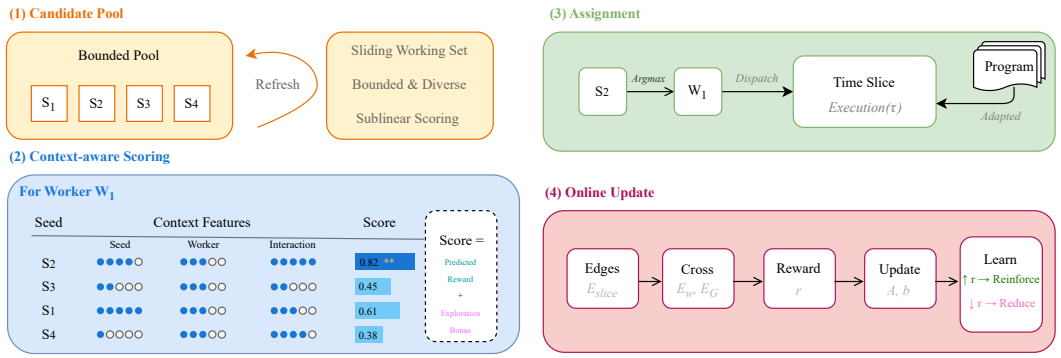


Fig. 3. **Contextual bandit-based scheduling in SimiFuzz.** The scheduler operates in four steps: maintaining a bounded candidate pool from the global corpus; computing context-aware scores for each seed–worker pair by combining predicted reward and exploration bonus; selecting the highest-scoring seed for each worker; collecting time-slice feedback and updating the model online. This closed-loop enables adaptive scheduling tailored to runtime fuzzing dynamics.

Candidate pool. Rather than scoring the entire global corpus at every decision, SimiFuzz maintains a bounded candidate pool C_t that serves as the action set at time t. Intuitively, C_t captures a diverse subset of seeds that are *available* to allocate and *worth considering* for learning. This design provides two practical benefits that are central to our model: it keeps the scoring cost sublinear in the corpus size, and it stabilizes online learning by avoiding frequent evaluation over a rapidly growing, highly redundant seed set. Conceptually, C_t can be viewed as a sliding working set populated by newly discovered seeds and periodically refreshed to preserve diversity. For each worker w, the scheduler evaluates candidate pairs (s, w) where $s \in C_t$.

LinUCB scoring. To balance exploitation and exploration, SimiFuzz adopts a linear upper-confidence estimator over the context vector. For each candidate pair (s, w) with context $x = x(s, w) \in \mathbb{R}^8$, the scheduler computes a confidence-aware score

$$\text{score}(x) \triangleq \hat{\mu}(x) + \alpha \cdot \hat{\sigma}(x), \tag{7}$$

where $\hat{\mu}(x)$ denotes the predicted reward and $\hat{\sigma}(x)$ denotes an uncertainty term. The parameter α controls how aggressively the scheduler explores uncertain contexts. The linear structure matches our feature design: seed properties, worker states, and pair interactions are captured explicitly and updated incrementally, allowing the model to generalize across unseen pairs.

We instantiate $\hat{\mu}$ and $\hat{\sigma}$ by maintaining a ridge-regularized linear model. Let $A \in \mathbb{R}^{d\times d}$ and $b \in \mathbb{R}^d$ with $d = 8$, initialized as $A = \lambda I$ and $b = \mathbf{0}$. At time t, the estimated weight is $\hat{\theta} = A^{-1}b$. The predicted mean is $\hat{\mu}(x) = \hat{\theta}^\top x$. The uncertainty is derived from the confidence ellipsoid induced by A: $\hat{\sigma}(x) = \sqrt{x^\top A^{-1} x}$. This estimator encourages exploration in rarely observed regions of the context space. This is important for fuzzing because worker states evolve over time, and a seed's utility depends on which worker executes it.

Assignment policy. For a given worker w, the scheduler selects the seed $s^*(w)$ and dispatches it for execution under the time-slice mechanism.

$$s^*(w) \in \arg\max_{s\in C_t} \text{score}\big(x(s, w)\big), \tag{8}$$

this policy operationalizes the core intuition behind SimiFuzz: a seed is not universally "good" or "bad"; instead, its expected benefit depends on the current worker state and the seed–worker

Algorithm 1 SimiFuzz online scheduling with time-slice feedback

Require: Candidate pool C_t; workers $\mathcal{W}$; ridge λ; exploration α; base budget τ_0

$A \leftarrow \lambda I;\ b \leftarrow \mathbf{0}$
Initialize global state E_G, per-worker state E_w, hot-edge sets H_w, EMA gains $m(w)$
while fuzzing is running **do**
  for all $w \in \mathcal{W}$ that becomes available **do**
    $\tau \leftarrow \textsc{AdjustBudget}(\tau_0,\ \text{avg\_exec}(w))$
    $best \leftarrow -\infty;\ s^\star \leftarrow \varnothing$
    for all $s \in C_t$ **do**
      $x \leftarrow \textsc{BuildContext}(s,\ w)$
      $u \leftarrow \text{UCBScore}(A,\ b,\ \alpha,\ x)$
      if $u > best$ **then**
        $best \leftarrow u;\ s^\star \leftarrow s$
      end if
    end for
    $\textsc{Dispatch}(s^\star,\ w,\ \tau)$
    $E_{\text{slice}} \leftarrow \textsc{CollectEdges}(w,\ \tau)$
    $r \leftarrow \textsc{ComputeReward}(E_{\text{slice}},\ E_G,\ E_w)$
    $x \leftarrow \textsc{BuildContext}(s^\star,\ w)$
    $A \leftarrow A + xx^\top;\ b \leftarrow b + rx$
    $\textsc{UpdateState}(E_{\text{slice}},\ E_G,\ E_w,\ H_w,\ m(w))$
    $\textsc{RefreshPool}(C_t)$
  end for
end while

interaction captured by $x^{\text{int}}(s, w)$. As a result, the scheduler can preferentially allocate seeds that either amplify a worker that is currently productive or diversify exploration by steering workers away from highly overlapping search regions.

Reward signal and online update. For each executed pair (s, w), the scheduler observes slice-level feedback and converts it into a bounded scalar reward r (§ 3.4). The reward is then attributed to the executed context $x(s, w)$ and used to update the LinUCB statistics with a lightweight rank-one update:

$$A \leftarrow A + xx^\top, \quad b \leftarrow b + rx. \tag{9}$$

This incremental update requires only the current context and a scalar reward, and does not depend on corpus-wide history. Over time, the model learns which context patterns tend to yield higher payoff, while the confidence term in Eq. (7) naturally shrinks for frequently sampled contexts.

End-to-end scheduling loop. Algorithm 1 summarizes the closed loop of SimiFuzz. At each decision point, the scheduler selects a seed from the bounded candidate pool C_t for a worker w by maximizing the contextual UCB score, dispatches the pair for a time slice, derives reward from the slice-level edge feedback, and updates the bandit model online. This coupling of a bounded candidate pool, contextual UCB scoring, and slice-derived reward enables SimiFuzz to adapt allocation decisions to non-stationary fuzzing dynamics while keeping decision overhead low.

3.4 Time-Slice Feedback Mechanism

To support frequent yet stable online learning, SimiFuzz adopts a time-slice feedback mechanism. Each scheduling decision assigns a seed–worker pair (s, w) for a bounded execution budget τ. Instead of attributing reward to individual executions, the worker repeatedly mutates and runs

the target within the slice and reports an aggregated edge summary at the end of the slice. This aggregation serves two purposes: it reduces the variance of short-horizon signals (single executions are highly noisy and heavy-tailed), and it provides a clear credit assignment that associates one reward value with one allocation action. We first specify the slice budget τ to amortize overhead, then derive slice-level edge feedback, construct a bounded reward, and finally use it for online model updates.

Dynamic budget to amortize overhead. A subtle but important consideration is that the effectiveness of time-slicing depends on the target's execution speed. With a fixed budget (e.g., $\tau = 100\,\text{ms}$), fast targets are more sensitive to per-slice scheduling overhead because a short boundary delay can replace many potential executions. To keep the overhead ratio below a small constant (our design target is $< 10\,\%$), we select τ such that the budget dominates scheduling time. Let T_s denote the per-slice scheduling overhead (including decision and synchronization). We require

$$\frac{T_s}{\tau + T_s} < 0.1 \iff \frac{\tau}{T_s} > 9. \tag{10}$$

Assuming T_s is on the order of a few tens of milliseconds, this implies that fast targets should receive budgets on the order of several hundred milliseconds to amortize overhead. In contrast, slow targets naturally amortize overhead even with smaller budgets, and thus can use shorter slices for finer-grained adaptation. Following this principle, SimiFuzz uses a coarse-grained budget policy that assigns larger τ to fast targets and smaller τ to slow targets, preserving both feedback efficiency and adaptivity across diverse workloads.

Slice-level edge feedback. Let $E_{\text{slice}}(s, w)$ denote the set (or bitmap) of edges hit during the slice when worker w executes mutations derived from seed s. From this slice summary, SimiFuzz derives incremental coverage statistics by comparing E_{slice} against the global covered set E_G and the worker-local covered set E_w. We classify each edge hit in the slice into three categories: *globally new* edges that are absent from E_G, *worker-new* edges that are already in E_G but absent from E_w, and *redundant* edges that are already covered by both E_G and E_w. Accordingly, we define the slice deltas

$$\Delta_{\text{global}} \triangleq \left|E_{\text{slice}} \setminus E_G\right|, \quad \Delta_{\text{cross}} \triangleq \left|\left(E_{\text{slice}} \cap E_G\right) \setminus E_w\right|, \quad \Delta_{\text{dup}} \triangleq \left|E_{\text{slice}} \cap E_w\right|. \tag{11}$$

where Δ_{global} captures immediate global progress, Δ_{cross} captures cross-worker knowledge transfer, and Δ_{dup} reflects redundant exploration within the worker's already-covered region. We further use Δ_{frontier} to measure how much the slice exercises the seed's birth-time frontier, namely the number of edges in $(E_s \setminus E_G^{\text{birth}})$ that also appear in E_{slice}, computed consistently with the frontier signal in § 3.2. Note that E_s denotes the seed-level hit-edge snapshot recorded when s is ingested (§ 3.2), whereas E_{slice} summarizes the edges exercised during the current time slice for reward and model update.

Reward construction. The reward should primarily reflect global exploration progress, while still crediting slice outcomes that improve collective efficiency over time. We therefore combine multiple signals into a bounded scalar reward:

$$r \triangleq \text{clip}\big(w_{\text{cov}} \cdot \phi(\Delta_{\text{global}}) + w_{\text{front}} \cdot \phi(\Delta_{\text{frontier}}) + w_{\text{cross}} \cdot \phi(\Delta_{\text{cross}}),\ r_{\min},\ r_{\max}\big)\,. \tag{12}$$

The saturating map $\phi(\cdot)$ prevents rare large deltas from dominating online learning, and the bounds $[r_{\min}, r_{\max}]$ improve numerical stability for incremental regression. The dominant term rewards globally new edges. The auxiliary terms capture two complementary effects. The frontier term credits seeds that expose globally novel behaviors when they are discovered, whereas the cross-learning term credits assignments that teach a worker behaviors already known to the group.

Cross-learning is particularly important in later stages. Many seeds no longer yield globally new edges, yet they can still be valuable for a worker that has not reached those regions. Consider

a seed s with $E_s \subseteq E_G$, which implies $\Delta_{\text{global}} \approx 0$. If worker w has not covered a large portion of E_s, the slice can still yield a large Δ_{cross}, effectively bringing w closer to the group's explored boundary. Rewarding this worker-new progress promotes knowledge transfer across workers, reduces repeated effort, and increases the chance that workers reach distinct frontiers in subsequent slices.

Online model update. After the slice finishes, the scheduler attributes the obtained reward r to the executed context $x(s, w)$ and performs an incremental update to the LinUCB statistics (Eq. 9). The slice-level aggregation makes these updates stable: repeated mutations within a slice average out stochastic effects, while the bounded reward prevents extreme outcomes from causing unstable parameter drift.

4 Implementation

We implement SimiFuzz as a layered system consisting of a Python scheduler, an AFL++ monitor, and multiple AFL++ workers. The scheduler is responsible for feature construction and LinUCB-based decision making, while the AFL++ side handles global seed management, task dispatch, and time-slice execution. Overall, our prototype contains approximately 4,800 lines of code, including about 2,000 lines of C/C++ modifications and around 2,800 lines of Python code. This split keeps the execution path close to AFL++ and confines learning and policy logic to Python for rapid iteration and portability.

Communication and control plane. The scheduler and AFL++ components communicate via Unix Domain Sockets (UDS). We implement six message types (Hello, Ingest, Credit, Report, Decide, and Assign) to complete the end-to-end loop from handshake and seed synchronization to request, dispatch, execution, and feedback. To reduce synchronization overhead, we aggregate execution feedback at the time-slice granularity: workers report a set of hit edges per slice via the Report message rather than per execution, which amortizes communication and decision latency across many executions. A monotonically increasing `dispatch_id` threads Decide→Assign→Report to align decisions with feedback and to mitigate noise from reordering or duplicates.

Feature construction and normalization. Each scheduling action corresponds to a *seed–worker* pair, represented by an 8-dimensional context vector. Ratio-based and set-based signals (e.g., coverage ratio, similarity, and intersection-derived signals) are already bounded and are used directly. For scale-sensitive seed properties (seed length and execution speed), we apply a Robust Z-score on log-transformed values with a variance floor and a soft clipping range. Concretely, we use a standard-deviation floor (`SIGMA_FLOOR=0.7`) to prevent instability when the corpus is homogeneous early in the run, and we clip Z-scores to ± 2 (`Z_MAX=2.0`) before mapping them to $[0, 1]$, which preserves gradient in the typical range while preventing extreme outliers from dominating decisions.

Candidate construction and scheduling overhead. We avoid enumerating all seed×worker combinations at each decision. Instead, we form a bounded candidate set and score only those pairs. The candidate-set size is controlled by `candidate_multiplier` (default 200), chosen to balance decision quality against online latency: it is large enough to cover recently active/high-potential seeds in the global pool, yet small enough to keep scoring cost stable in Python. Increasing the multiplier beyond this point yields diminishing returns while linearly increasing per-decision time and waiting overhead.

Stabilizing LinUCB. We initialize LinUCB with ridge regularization `ridge_l2=0.001` (i.e., $A = \lambda I$) to improve numerical stability without overly slowing early learning. We set the exploration coefficient to `alpha=0.05`, calibrated to the reward scale in our implementation (see below): this value makes uncertainty matter during cold start while avoiding persistent oscillations driven by over-exploration.

Reward shaping and clipping. We derive a scalar reward from each time slice by classifying edges hit in that slice into global-new, cross-learning, and duplicate categories. The weight trade-off is governed by `gamma=0.3`, which biases the objective toward coverage while reserving sufficient mass for long-term potential and cross-worker learning signals. In preliminary runs, larger γ tended to over-optimize short-term gains and concentrate effort, whereas smaller γ weakened the primary coverage objective. We further clip rewards to `REWARD_FLOOR=0.01` and `REWARD_CAP=0.2`: the floor avoids zero-signal updates (important for online learning), and the cap prevents rare extreme slices from producing unstable parameter jumps.

Overlap penalty and temporal windowing. To discourage redundant exploration among workers, we apply a mild overlap penalty with `lambda_overlap=0.1`, which is sufficient to break ties among otherwise similar candidates without overwhelming learned preferences. Similarity and "hot-edge" statistics are computed over a sliding window controlled by `window_secs=600`. A shorter window made the signal noisy at time-slice granularity, while a much longer window diluted recency; 10 minutes provided a robust compromise between responsiveness and stability in our setting.

Time-slice budget. We set the default time-slice length to `budget_ms=100`, which provides frequent feedback and timely policy updates. For fast-executing targets, we increase the budget to amortize communication and decision overhead; for slower targets, we keep the default to maintain responsiveness. This simple adaptation keeps the scheduling/communication fraction within a practical range without sacrificing the ability to react to changing worker states.

5 Evaluation

We conduct a comprehensive evaluation of SimiFuzz to answer the following questions.

- **RQ1:** How does SimiFuzz perform on code coverage compared to existing parallel fuzzers?
- **RQ2:** How effective is SimiFuzz at bug/vulnerability detection?
- **RQ3:** What are the contributions of the main components of SimiFuzz?

5.1 Experimental Setup

Environment. We evaluate SimiFuzz on six identical machines. Each machine is equipped with dual AMD EPYC 7K62 48-core CPUs (96 hardware threads per CPU) and 64 GB RAM. All experiments were conducted in Docker containers on Ubuntu 20.04. Unless otherwise specified, each run uses 10 CPU cores for 24 hours, and we repeat every configuration for 10 independent trials with the same initial seeds.

Baselines. We compare SimiFuzz with four representative parallel fuzzers:

- **AFL++ (parallel mode)** runs multiple synchronized instances (one master and multiple slaves) and periodically exchanges discovered seeds via the sync mechanism.
- **PAFL** is selected as a primary baseline due to its static task allocation on bitmap partitions. PAFL was originally implemented on AFL; we port it to AFL++ to align instrumentation and runtime settings for a fair comparison.
- **FlexFuzz** represents heuristic-based dynamic allocation with structure-aware scheduling, and serves as a strong baseline for adaptive task partitioning.
- **KRAKEN** employs an Ant Colony Optimization (ACO) strategy to adaptively adjust parallel allocation, representing optimization-driven dynamic scheduling.

We exclude DynamiQ because its implementation is not publicly available, and exclude μFUZZ due to persistent runtime memory errors in our setup.

Benchmark. We evaluate all fuzzers on eight widely used real-world targets drawn from four sources: FuzzBench [27], OSS-Fuzz [31], UniFuzz [19], and production software commonly used

Table 2. Target programs evaluated in the experiments.

Binary	Project	Version	Type	Command-line	Size (KiB)
pngfix	Libpng	1.6.34	Image	–strip=color –out=/dev/null @@	1,207
tcpdump	Tcpdump	4.8.1	Network	-e -vv -nr @@	3,494
sqlite3	SQLite	3.8.9	Database	<	4,538
xmllint	Libxml2	2.10.4	XML	–valid @@	6,233
pdftotext	Xpdf	4.00	PDF	@@ /dev/null	7,033
objdump	Binutils	2.28	Binary	-S @@	9,514
exiv2	Exiv2	0.26	Image	@@	16,677
ffmpeg	FFmpeg	4.0.1	Video	-y -i @@ -c:v mpeg4 -c:a copy -f mp4 /dev/null	88,208

Table 3. The average number of covered edges across ten trials and the relative improvement of SimiFuzz over each baseline.

Program	AFL++		PAFL		KRAKEN		FlexFuzz		SimiFuzz
	Number	Imp.	Number	Imp.	Number	Imp.	Number	Imp.	
libpng	1,210.80	0.68 %	1,207.90	0.92 %	1,212.20†	0.56 %†	1,181.00	3.22 %	**1,219.00**
tcpdump	9,704.70	26.96 %	8,421.80	46.30 %	7,523.50	63.77 %	10,705.40†	15.10 %†	**12,321.40**
sqlite3	9,510.50†	23.05 %†	8,422.20	38.96 %	7,802.60	49.99 %	8,612.70	35.88 %	**11,703.10**
libxml2	5,201.50	25.25 %	4,851.80	34.27 %	5,255.30	23.96 %	5,881.40†	10.77 %†	**6,514.70**
pdftotext	7,644.90	23.44 %	7,268.30	29.83 %	6,311.50	49.51 %	7,961.90†	18.52 %†	**9,436.50**
objdump	4,477.60	26.03 %	4,198.30	34.42 %	3,718.60	51.76 %	5,104.10†	10.56 %†	**5,643.30**
exiv2	7,680.50	27.89 %	7,713.50	27.34 %	7,061.50	39.10 %	8,941.20†	9.86 %†	**9,822.60**
ffmpeg	36,231.70	34.01 %	27,941.40	73.78 %	16,367.50	196.66 %	45,756.90†	6.12 %†	**48,555.90**
AVG	10,207.78	28.84 %	8,753.15	50.26 %	6,906.59	90.43 %	11,768.08†	11.76 %†	**13,152.06**

Bold and † denote the best and second-best coverage results, respectively.
Imp. denotes the relative improvement of SimiFuzz over the corresponding baseline, computed as $(Cov_{\text{SimiFuzz}} - Cov_{\text{baseline}})/Cov_{\text{baseline}} \times 100\,\%$.

in vulnerability research. Table 2 summarizes the detailed benchmark configuration, including target versions, input domains, command-line invocations, and binary sizes. The target selection is guided by four factors: diversity of input/file-processing workloads, broad real-world adoption, program scale, and suitability for parallel fuzzing. Because smaller programs often benefit little from parallel fuzzing, we favor targets with sufficient scale and exploration complexity to make parallel scheduling decisions non-trivial. The final set spans packet and document processing (`tcpdump`, `pdftotext`), media and image handling (`ffmpeg`, `libpng`, `exiv2`), data management (`sqlite3`), and binary utilities/parsers (`objdump`, `libxml2`). These targets expose heterogeneous execution costs and exploration behaviors, thereby stressing both the utility model and the diversity-aware scheduler. Unless otherwise stated, we adopt the standard fuzzing harnesses provided by the benchmark suites or upstream fuzzing integrations, and compile every target with identical compiler, instrumentation, and sanitizer settings across all baselines.

5.2 RQ1: Code Coverage

We measure effectiveness by the number of distinct covered edges after 24 hours. All fuzzers were executed with 10 cores in parallel on each target, and each configuration was repeated for ten independent trials. Since KRAKEN and FlexFuzz use different instrumentation, we report their

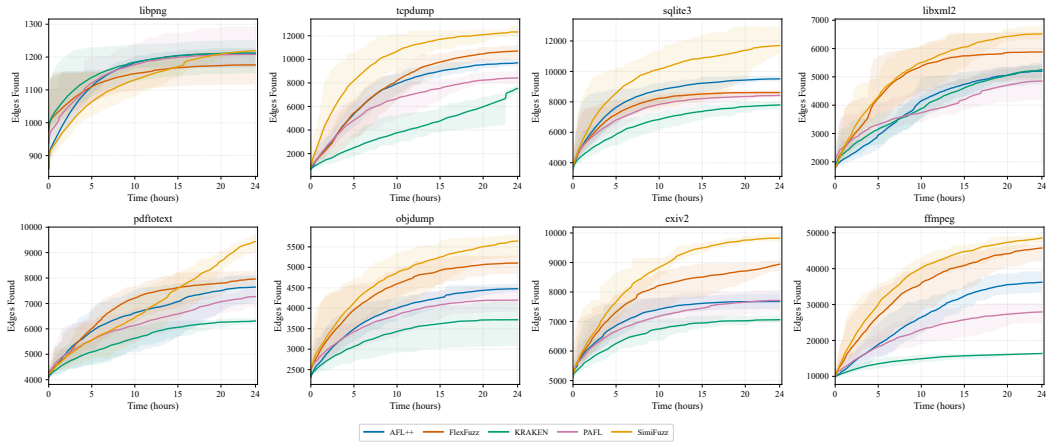


Fig. 4. Edge-coverage trends over 24 hours across benchmark targets. Each line denotes the mean covered edges over ten trials, and the shaded band denotes the min–max range. Figure 7 follows the same convention.

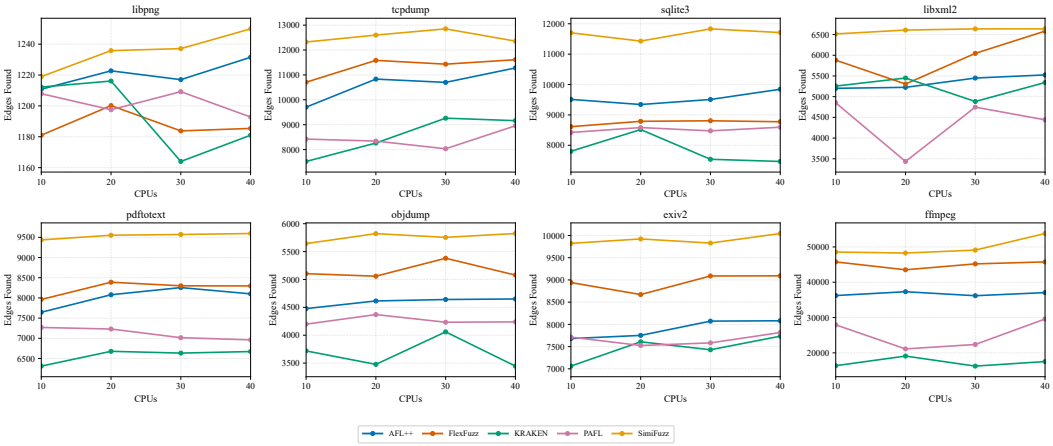


Fig. 5. Scalability of average final edge coverage under different CPU-core settings over ten trials.

edge coverage by replaying their generated inputs on AFL++-instrumented binaries to obtain a comparable metric. Concretely, we use `afl-cmin` to replay each fuzzer's generated queue on the same AFL++-instrumented binary and collect the resulting edge coverage. Table 3 reports the mean final coverage, and Figure 4 shows the coverage trajectory for each target, where each line denotes the mean over ten trials and the surrounding shaded band denotes the min–max range across trials.

SimiFuzz achieves the highest final coverage on all eight targets. Averaged across targets, it improves final coverage by 28.84 % over AFL++, 50.26 % over PAFL, 90.43 % over KRAKEN, and 11.76 % over FlexFuzz, which is the strongest coverage baseline overall. In particular, SimiFuzz increases the average final coverage from 11,768.08 edges with FlexFuzz to 13,152.06 edges.

The gains are most pronounced on large and exploration-intensive targets. On `ffmpeg`, SimiFuzz reaches 48,555.90 edges, outperforming FlexFuzz by 6.12 % and AFL++ by 34.01 %. It also substantially exceeds FlexFuzz on `tcpdump` (+15.10 %), `pdftotext` (+18.52 %), `sqlite3` (+35.88 %), `objdump` (+10.56 %), `exiv2` (+9.86 %), and `libxml2` (+10.77 %). On `sqlite3`, where AFL++ is the

Table 4. Vulnerability and CVE detection across ten trials.

Program	AFL++		PAFL		KRAKEN		FlexFuzz		SimiFuzz	
	VUL	CVE	VUL	CVE	VUL	CVE	VUL	CVE	VUL	CVE
sqlite3	14†	1†	11	0	12	0	11	0	**16**	**1**
pdftotext	20	11	24	11	23	12	27†	12†	**30**	**15**
objdump	7	4	9	4	6	3	10†	4†	**14**	**6**
exiv2	15	9†	15	7	15	6	18†	7	**21**	**11**
ffmpeg	3	1	2	1	2	1	6†	1†	**7**	**2**
SUM	59	26†	61	23	58	22	72†	24	**88**	**35**

Bold and † denote the best and second-best results, respectively.

strongest baseline on final coverage, SimiFuzz still improves over it by 23.05 %, suggesting stronger robustness across targets.

Figure 4 further indicates that SimiFuzz often accelerates mean coverage growth, establishing an early lead on several targets (e.g., `tcpdump`, `ffmpeg`, `libxml2`) and maintaining it throughout the run. On some targets, however, the advantage becomes clear only after more feedback has accumulated and worker states have further diverged. This pattern is visible on `pdftotext` and `libpng`, where SimiFuzz is slightly behind in the early stage but overtakes the baselines later and still achieves the best final coverage. This cold-start behavior is a limitation of learning-based scheduling: before sufficient feedback is collected, simpler heuristics may occasionally make faster early progress. For near-saturated targets such as `libpng`, all fuzzers converge quickly and the remaining headroom is small, consistent with the limited differences in final coverage.

To further evaluate scalability, Figure 5 compares the average final edge coverage over ten trials under 10, 20, 30, and 40 CPU cores. SimiFuzz consistently achieves the highest average coverage across the tested core settings on all benchmark targets. Although the average coverage does not always increase monotonically with more cores, mainly because additional workers can also introduce redundant exploration and scheduling variance, SimiFuzz maintains a stable advantage over the baselines. These results suggest that the seed–worker affinity model continues to provide useful scheduling guidance as the degree of parallelism increases in the multicore setting.

5.3 RQ2: Bug Detection

We assess bug-finding capability via a sanitizer-driven triage pipeline with conservative de-duplication. Targets are compiled with Clang sanitizers, and we primarily rely on AddressSanitizer (ASan) reports to identify memory-safety violations [22, 32]. For each run, we collect all crashing inputs and group them by sanitizer type and stack-trace signature to collapse trivial duplicates. Since a fuzzing campaign can still yield a large number of near-identical crashes, we adopt the *top-3* stack hashing method from prior fuzzing evaluations [14]: using the top three code locations in the ASan stack trace as unique crash signatures. We then manually inspect the remaining reproducible reports across trials to further merge duplicates that correspond to the same root cause, yielding the number of distinct *vulnerabilities* (VUL).

For known issues, we additionally map a subset of VULs to CVEs by matching our reproduced crash evidence against the corresponding entries on the official CVE database and validating that the trigger agrees with the public description. VUL therefore includes both CVE-matched cases and additional issues that cannot be mapped to a CVE, e.g., vulnerabilities without an assigned identifier or low-impact findings unlikely to receive a CVE.

Table 5. Vulnerability-type distribution across fuzzers.

Fuzzer	SEGV	HBO	GBO	BL	HUAF	BMUS	SO	ADM	FPE	ASTB	OOM	Tot
AFL++	19 (s5,p7,o3,e3,f1)	22 (s4,p8,o1,e8,f1)	3 (s1,p1,e1)	1 (s1)	2 (s2)	1 (s1)	4 (p2,o1,e1)	1 (p1)	4 (p1,o1,e1,f1)	1 (e1)	1 (o1)	59 (s14,p20,o7,e15,f3)
PAFL	17 (s3,p6,o4,e4)	21 (s3,p9,o2,e6,f1)	3 (s1,p1,e1)	1 (s1)	3 (s3)	0 (-)	9 (p7,o1,e1)	0 (-)	5 (p1,o1,e2,f1)	1 (e1)	1 (o1)	61 (s11,p24,o9,e15,f2)
KRAKEN	19 (s5,p7,o2,e4,f1)	18 (s2,p9,o1,e5,f1)	3 (s1,p1,e1)	1 (s1)	3 (s3)	0 (-)	8 (p5,o1,e2)	0 (-)	3 (p1,o1,e1)	2 (e2)	1 (o1)	58 (s12,p23,o6,e15,f2)
FlexFuzz	22 (s3,p8,o4,e4,f3)	21 (s2,p8,o3,e7,f1)	4 (s1,p2,e1)	1 (s1)	3 (s3)	1 (s1)	10 (p7,o1,e2)	1 (p1)	6 (p1,o1,e2,f2)	2 (e2)	1 (o1)	72 (s11,p27,o10,e18,f6)
SimiFuzz	27 (s5,p10,o5,e4,f3)	30 (s4,p9,o5,e10,f2)	4 (s1,p2,e1)	1 (s1)	4 (s4)	1 (s1)	10 (p7,o1,e2)	1 (p1)	6 (p1,o1,e2,f2)	3 (o1,e2)	1 (o1)	88 (s16,p30,o14,e21,f7)

Each entry additionally reports the per-target distribution in parentheses, where s = sqlite3, p = pdftotext, o = objdump, e = exiv2, and f = ffmpeg. Abbreviations: SEGV = segmentation fault, HBO = heap-buffer-overflow, GBO = global-buffer-overflow, BL = bytes_leaked, HUAF = heap-use-after-free, BMUS = bad-malloc_usable_size, SO = stack-overflow, ADM = alloc-dealloc-mismatch, FPE = floating-point exception, ASTB = allocation-size-too-big, OOM = out-of-memory, and Tot = total.

Table 4 reports the aggregated results over ten trials. SimiFuzz finds 88 distinct vulnerabilities and maps 35 of them to CVEs, which is the best among all evaluated fuzzers. FlexFuzz ranks second in VULs with 72 vulnerabilities and 24 CVEs, while AFL++ reports 59 vulnerabilities and the second-highest number of CVEs with 26. The advantage of SimiFuzz is visible on every target in this analysis set: it detects 30 vulnerabilities and 15 CVEs on `pdftotext`, 21 and 11 on `exiv2`, 14 and 6 on `objdump`, and 7 and 2 on `ffmpeg`. Figure 6 shows that SimiFuzz reproduces all CVEs that are shared by multiple fuzzers and additionally reproduces 7 CVEs that are unique to SimiFuzz in our setup, indicating broader bug-finding coverage beyond the shared overlap.

Table 5 shows the vulnerability-type distribution of the distinct VULs found by each fuzzer. Overall, the advantage of SimiFuzz does not appear to come from a single failure mode alone. Its gains are more visible on several common categories, especially segmentation faults and heap-buffer-overflows, while on the remaining categories it is generally competitive with the strongest baseline. This pattern suggests that the benefit of SimiFuzz is not limited to one specific bug class, but may reflect a broader bug-finding capability across different targets and vulnerability types.

5.4 RQ3: Ablation Study

This ablation study isolates the effects of interaction-aware context modeling and cross-learning reward design. We compare full SimiFuzz with three variants: SimiFuzz $_{\text{w/o-int}}$, which removes all interaction features $x^{\text{int}}(s, w)$; SimiFuzz $_{\text{w/o-cross}}$, which removes only the cross feature $\text{cross}(s, w)$; and SimiFuzz $_{\text{w/o-cl-reward}}$, which keeps the full context but disables the cross-learning term in Eq. 12 by setting $w_{\text{cross}} = 0$. Table 6 reports the final edge coverage and bug-finding results over ten trials.

The results show that each component contributes to the overall effectiveness of SimiFuzz, as removing them leads to different degrees of degradation. Removing all interaction features causes the largest decrease: SimiFuzz $_{\text{w/o-int}}$ loses 1,705.29 edges (12.97 %) on average and finds 21 fewer VULs and 11 fewer CVEs than full SimiFuzz, showing that interaction-aware context is the main source of improvement. Removing only the cross feature $\text{cross}(s, w)$ leads to a smaller but still clear decrease of 417.94 edges (3.18 %) on average and finds 9 fewer VULs and 2 fewer CVEs than full SimiFuzz, indicating that cross-worker coverage awareness is an important part of the interaction representation. Disabling the reward-side cross-learning term results in the smallest decrease, 274.11 edges (2.08 %) on average and finds 4 fewer VULs and 1 fewer CVE than full SimiFuzz, suggesting that this reward signal provides an additional coordination benefit.

Table 6. Ablation results over ten trials.

Program	SimiFuzz$_{w/o-int}$				SimiFuzz$_{w/o-cross}$				SimiFuzz$_{w/o-cl-reward}$			
	Number	Δ vs Full	VUL	CVE	Number	Δ vs Full	VUL	CVE	Number	Δ vs Full	VUL	CVE
libpng	1,214.00	-0.41 %	0	0	1,219.20	+0.02 %	0	0	1,216.60	-0.20 %	0	0
tcpdump	10,749.40	-12.76 %	0	0	11,938.40	-3.11 %	0	0	12,127.10	-1.58 %	0	0
sqlite3	10,424.30	-10.93 %	13	0	11,562.40	-1.20 %	15	1	11,432.10	-2.32 %	15	1
libxml2	5,755.80	-11.65 %	0	0	6,248.40	-4.09 %	0	0	6,372.50	-2.18 %	0	0
pdftotext	8,191.40	-13.19 %	23	12	8,956.40	-5.09 %	28	14	9,174.70	-2.77 %	29	14
objdump	4,947.20	-12.33 %	9	4	5,451.50	-3.40 %	11	6	5,520.80	-2.17 %	13	6
exiv2	8,587.60	-12.57 %	18	7	9,582.30	-2.45 %	19	10	9,658.50	-1.67 %	20	11
ffmpeg	41,704.50	-14.11 %	4	1	46,914.40	-3.38 %	6	2	47,521.30	-2.13 %	7	2
AVG/SUM	11,446.78	-12.97 %	67	24	12,734.13	-3.18 %	79	33	12,877.95	-2.08 %	84	34

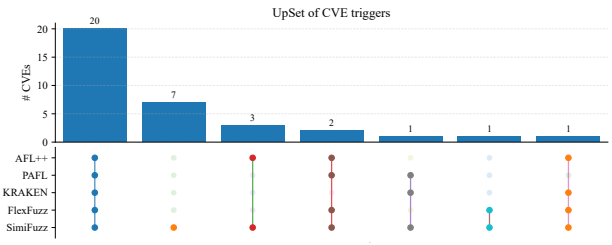


Fig. 6. UpSet plot of CVE reproducibility across fuzzers.

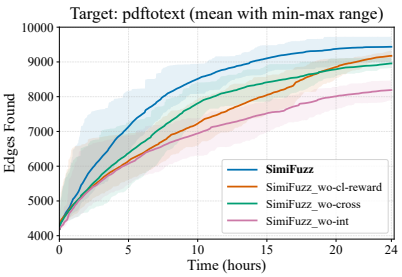


Fig. 7. Coverage trends on pdftotext over ten trials.

Figure 7 shows the same pattern on *pdftotext*. Full SimiFuzz maintains the strongest trajectory throughout the campaign, SimiFuzz $_{\text{w/o-int}}$ falls furthest behind, and SimiFuzz $_{\text{w/o-cross}}$ and SimiFuzz $_{\text{w/o-cl-reward}}$ stay in between. By the end of the campaign, these three variants trail full SimiFuzz by 1,245.10, 480.10, and 261.80 edges, respectively. Overall, the ablation results indicate that the main gain of SimiFuzz comes from interaction-aware representation, while cross-learning reward shaping further improves coordination over time.

6 Related Work

6.1 Parallel Fuzzing and Task Allocation

Parallel fuzzing commonly starts from the master–slave style replication adopted by AFL/AFL++ [11, 40], and then improves efficiency by reducing redundant exploration. One line of work performs static task partitioning: PAFL extends single-instance optimizations to parallel mode [21]. Another line focuses on coordination: EnFuzz exchanges seeds among diverse fuzzers [7] and AFL-EDGE distributes mutually-exclusive tasks [36], while CollabFuzz shares information across participants [28]. Dynamic schedulers allocate work based on runtime signals; AFLTeam performs systematic task allocation [29], and FlexFuzz schedules workers with boundary-targeted exploration [20]. Learning-based approaches further automate allocation; KRAKEN adapts parallel fuzzing with a learning-driven strategy [41]. Meta-level approaches such as autofz dynamically compose and reallocate resources across fuzzers [12].

6.2 Learning-Guided Fuzzing

Machine learning has been applied to improve fuzzing efficiency in seed scheduling and resource control. AFLFast introduces power schedules based on Markov chain modeling [5], and AFLGo extends this with simulated annealing for directed fuzzing [4]. SLIME proposes program-sensitive energy allocation [25], and FairFuzz targets rare branches with adaptive mutation masks [16]. EcoFuzz formulates energy allocation as an adversarial bandit problem [39], while AFL-HIER uses reinforcement learning for hierarchical seed scheduling [35]. T-Scheduler applies Thompson Sampling with theoretical guarantees [23]. For kernel fuzzing, SyzVegas employs multi-armed bandits for joint task and seed selection [34]. Complementary to seed scheduling, MOpt optimizes mutation probabilities [24], and HavocMAB applies bandit optimization to the havoc stage [37]. The multi-armed bandit (MAB) problem provides a principled framework for exploration-exploitation trade-offs [2, 3, 15, 30]. Contextual MAB incorporates side information [1, 6, 9, 18]. However, these approaches treat each seed independently without modeling seed–worker interactions. Our work applies contextual bandits to parallel fuzzing, jointly modeling seed properties, worker states, and their interactions for adaptive scheduling.

7 Conclusion

We have presented SimiFuzz, a context-aware scheduling framework for parallel fuzzing that learns seed–worker assignments from time-slice feedback. Across eight real-world targets, SimiFuzz improves edge coverage and bug-finding results over AFL++ and other strong parallel baselines. A limitation is that our scheduler is still driven by AFL-style edge feedback and a linear reward signal, which may miss richer semantics in complex programs. In addition, SimiFuzz requires feedback to learn reliable seed–worker affinities, so its benefit may be less pronounced during the cold-start phase of a campaign. In future work, we plan to explore hybrid warm-start strategies that begin with lightweight heuristic scheduling and gradually shift to learned seed–worker assignment as feedback accumulates. We also plan to introduce large language models as a decision layer to replace RL-style training, leveraging structured execution summaries to make more informed and interpretable scheduling choices.

Data Availability

We will open source SimiFuzz at https://github.com/De3mond01/SimiFuzz, and will also include evaluation data in §5.2, §5.3, §5.4.

Acknowledgments

This research was supported by the Zhejiang Provincial Natural Science Foundation of China (Grant Nos. LZYQ25F020003, LD24F020002, and LQ24F020025); the Key Project of the National Natural Science Foundation of China (Grant No. 62536007); the National Key Research and Development Program of China (Grant No. 2023YFB3107100); the National Natural Science Foundation of China (Grant No. 62502454); the Major Key Project of PCL (Grant No. PCL2024A05); Zhejiang Province's 2025 "Leading Goose + X" Science and Technology Plan (Grant No. 2025C02034); and the Jinhua Science and Technology Plan (Grant Nos. 2023-1-091 and 2026-1-001). We thank the anonymous reviewers for their constructive comments, which helped us improve the paper.

References

[1] Yasin Abbasi-Yadkori, Dávid Pál, and Csaba Szepesvári. 2011. Improved algorithms for linear stochastic bandits. In *Advances in Neural Information Processing Systems*, Vol. 24. Curran Associates, Inc., 2312–2320.

[2] Rajeev Agrawal. 1995. Sample mean based index policies by O(log n) regret for the multi-armed bandit problem. *Advances in Applied Probability* 27, 4 (1995), 1054–1078. doi:10.2307/1427934

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

[4] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2329–2344. doi:10.1145/3133956.3134020

[5] 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 (CCS)*. ACM, 1032–1043. doi:10.1145/2976749.2978428

[6] Olivier Chapelle and Lihong Li. 2011. An empirical evaluation of Thompson sampling. In *Advances in Neural Information Processing Systems*, Vol. 24. Curran Associates, Inc., 2249–2257.

[7] 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 *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 1967–1983.

[8] Yongheng Chen, Rui Zhong, Yupeng Yang, Hong Hu, Dinghao Wu, and Wenke Lee. 2023. μFUZZ: Redesign of Parallel Fuzzing Using Microservice Architecture. In *32nd USENIX Security Symposium (USENIX Security'23)*. USENIX Association, 1325–1342.

[9] Wei Chu, Lihong Li, Lev Reyzin, and Robert Schapire. 2011. Contextual Bandits with Linear Payoff Functions. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 15)*. PMLR, 208–214.

[10] Zhen Yu Ding and Claire Le Goues. 2021. An Empirical Study of OSS-Fuzz Bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 131–142. doi:10.1109/MSR52588.2021.00026

[11] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, 12 pages. https://www.usenix.org/conference/woot20/presentation/fioraldi

[12] 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, 1901–1918. https://www.usenix.org/conference/usenixsecurity23/presentation/fu-yu-fu

[13] Taotao Gu, Xiang Li, Shuaibing Lu, Jianwen Tian, Yuanping Nie, Xiaohui Kuang, Zhechao Lin, Chenyifan Liu, Jie Liang, and Yu Jiang. 2022. Group-Based Corpus Scheduling for Parallel Fuzzing. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 1521–1532. doi:10.1145/3540250.3560885

[14] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2123–2138. doi:10.1145/3243734.3243804

[15] Tze Leung Lai and Herbert Robbins. 1985. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics* 6, 1 (1985), 4–22. doi:10.1016/0196-8858(85)90002-8

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

[17] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity* 1, 1, Article 6 (2018), 13 pages. doi:10.1186/s42400-018-0002-y

[18] Lihong Li, Wei Chu, John Langford, and Robert E. Schapire. 2010. A Contextual-Bandit Approach to Personalized News Article Recommendation. In *19th International Conference on World Wide Web (WWW)*. ACM, 661–670. doi:10.1145/1772690.1772758

[19] 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 *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2777–2794.

[20] Hong Liang, Yijia Guo, Haotian Wu, Yifan Xia, Yi Xiang, Xiantao Jin, Hao Peng, Xuhong Zhang, and Shouling Ji. 2025. Boosting Parallel Fuzzing with Boundary-Targeted Task Allocation and Exploration. *IEEE Transactions on Information Forensics and Security* 20 (2025), 11728–11743. doi:10.1109/TIFS.2025.3622956

[21] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. 2018. PAFL: Extend Fuzzing Optimizations of Single Mode to Industrial Parallel Mode. In *26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 809–814. doi:10.1145/3236024.3275525

[22] LLVM Project. 2026. *Sanitizers*. Clang/LLVM sanitizer documentation collection; tools originated in 2011–2012. Retrieved July 10, 2026 from https://clang.llvm.org/docs/index.html

[23] Simon Luo, Adrian Herrera, Paul Quirk, Michael Chase, Damith C. Ranasinghe, and Salil S. Kanhere. 2024. Make Out Like a (Multi-Armed) Bandit: Improving the Odds of Fuzzer Seed Scheduling with T-Scheduler. In *19th ACM ASIA Conference on Computer and Communications Security (AsiaCCS)*. ACM, 1463–1479. doi:10.1145/3634737.3637639

[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, 1949–1966.

[25] Chenyang Lyu, Hong Liang, Shouling Ji, Xuhong Zhang, Binbin Zhao, Meng Han, Yun Li, Zhe Wang, Wenhai Wang, and Raheem Beyah. 2022. SLIME: Program-Sensitive Energy Allocation for Fuzzing. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 365–377. doi:10.1145/3533767.3534385

[26] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2312–2331. doi:10.1109/TSE.2019.2946563

[27] 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 European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 1393–1403. doi:10.1145/3468264.3473932

[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 *14th European Workshop on Systems Security (EuroSec)*. ACM, 1–7. doi:10.1145/3447852.3458720

[29] Van-Thuan Pham, Manh-Dung Nguyen, Quang-Trung Ta, Toby Murray, and Benjamin I.P. Rubinstein. 2021. Towards Systematic and Dynamic Task Allocation for Collaborative Parallel Fuzzing. In *36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1337–1341. doi:10.1109/ASE51524.2021.9678810

[30] Herbert Robbins. 1952. Some aspects of the sequential design of experiments. *Bull. Amer. Math. Soc.* 58, 5 (1952), 527–535. doi:10.1090/s0002-9904-1952-09620-8

[31] Kostya Serebryany. 2017. OSS-Fuzz: Google's Continuous Fuzzing Service for Open Source Software. Talk at the 26th USENIX Security Symposium. Retrieved July 10, 2026 from https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany

[32] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, 309–318.

[33] Congxi Song, Xu Zhou, Qidi Yin, Xinglu He, Hangwei Zhang, and Kai Lu. 2019. P-Fuzz: A Parallel Grey-Box Fuzzing Framework. *Applied Sciences* 9, 23, Article 5100 (2019), 14 pages. doi:10.3390/app9235100

[34] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael Abu-Ghazaleh. 2021. SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2741–2758.

[35] Jinghan Wang, Chengyu Song, and Heng Yin. 2021. Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing. In *28th Annual Network and Distributed System Security Symposium (NDSS)*. Internet Society, 17 pages. doi:10.14722/ndss.2021.24486

[36] Yifan Wang, Yuchen Zhang, Chenbin Pang, Peng Li, Nikolaos Triandopoulos, and Jun Xu. 2021. Facilitating Parallel Fuzzing with Mutually-Exclusive Task Distribution. In *17th EAI International Conference on Security and Privacy in Communication Networks (SecureComm)*. Springer, 185–206. doi:10.1007/978-3-030-90022-9_10

[37] 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 (ICSE)*. ACM, 1634–1645. doi:10.1145/3510003.3510174

[38] Wenqi Yan, Toby Murray, Benjamin I. P. Rubinstein, and Van-Thuan Pham. 2025. *DynamiQ: Unlocking the Potential of Dynamic Task Allocation in Parallel Fuzzing*. arXiv:2510.04469 doi:10.48550/arXiv.2510.04469

[39] 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.

[40] Michal Zalewski. 2014. *American Fuzzy Lop (AFL)*. Retrieved July 10, 2026 from https://lcamtuf.coredump.cx/afl/

[41] Anshunkang Zhou, Heqing Huang, and Charles Zhang. 2025. KRAKEN: Program-Adaptive Parallel Fuzzing. *Proceedings of the ACM on Software Engineering* 2, ISSTA, Article ISSTA013 (July 2025), 23 pages. doi:10.1145/3728882

[42] Xu Zhou, Pengfei Wang, Chenyifan Liu, Tai Yue, Yingying Liu, Congxi Song, Kai Lu, Qidi Yin, and Xu Han. 2023. UltraFuzz: Towards Resource-Saving in Distributed Fuzzing. *IEEE Transactions on Software Engineering* 49, 4 (2023), 2394–2412. doi:10.1109/TSE.2022.3219520

Received 2026-01-30; accepted 2026-06-25