

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

TrioFuzz: A Three-Tier Architecture for Adaptive Strategy Selection in Fuzzing

Ruiqi Dong, *Swinburne University of Technology;* Yiyi Wang, *Tsinghua University and Zhongguancun Laboratory;* Kunpeng Zhang, *The Hong Kong University of Science and Technology;* Dongsong Yu, *Zhongguancun Laboratory;* Xiaogang Zhu, *Adelaide University;* Shaohua Wang, *Central University of Finance and Economics;* Shuai Wang, *The Hong Kong University of Science and Technology;* Chao Zhang, *Tsinghua University and Zhongguancun Laboratory;* Sheng Wen and Yang Xiang, *Swinburne University of Technology*

https://www.usenix.org/conference/usenixsecurity26/presentation/dong-ruiqi

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

TRIOFUZZ: A Three-Tier Architecture for Adaptive Strategy Selection in Fuzzing

Ruiqi Dong[1], Yiyi Wang[2,3], Kunpeng Zhang[4], Dongsong Yu[3], Xiaogang Zhu[5],
Shaohua Wang[6], Shuai Wang[4], Chao Zhang[2,3], Sheng Wen[1], Yang Xiang[1]
[1]*Swinburne University of Technology,* [2]*Tsinghua University,* [3]*Zhongguancun Laboratory*
[4]*The Hong Kong University of Science and Technology,* [5]*Adelaide University*
[6]*Central University of Finance and Economics*

Abstract

Adaptive strategy selection is a promising direction for improving fuzzing effectiveness, yet existing learning-based approaches often fail to outperform random selection in practice. We identify the root cause as an architectural mismatch rather than an algorithmic limitation. Our empirical study reveals that strategy effectiveness shifts at minute-level timescales, while existing in-loop architectures require tens of minutes per learning cycle. This order-of-magnitude gap prevents timely adaptation regardless of which learning algorithm is used.

We propose TRIOFUZZ, a new fuzzing engine that decouples learning from execution through a three-tier thread architecture. A centralized learning thread aggregates feedback from parallel execution threads, compressing the adaptation cycle to under one minute. TRIOFUZZ integrates the same learning algorithms as prior work, isolating the architectural contribution from algorithmic factors. Our ablation study shows that learning algorithms perform significantly better under our proposed architecture than under their original in-loop implementations. Notably, native MOpt underperforms baselines, but becomes a competitive performer under TRIOFUZZ's architecture. Evaluation on FuzzBench and OSS-Fuzz shows that TRIOFUZZ achieves the highest coverage on 7 out of 9 FuzzBench targets and 8 out of 10 OSS-Fuzz projects, triggers 17% more vulnerabilities than AFL++ on Magma, and discovers 19 new CVEs in 6 real-world projects.

1 Introduction

Coverage-guided greybox fuzzing has become one of the most effective techniques for automated software testing [51]. Modern fuzzers such as AFL++ [9], LibFuzzer [28], and Honggfuzz [13] employ diverse strategies from individual mutation operators (e.g., bit flips, arithmetic) to high-level algorithm configurations (e.g., Havoc, MOpt, CmpLog). AFL++'s havoc excels at broad random exploration. Redqueen [2] leverages input-to-state correspondence to solve magic byte comparisons. CmpLog [9] uses comparison operand feedback to guide targeted mutations. Large-scale empirical studies [22, 24, 31] have demonstrated that these strategies exhibit strong complementarity, as no single strategy dominates across all targets or execution phases. This complementarity has motivated significant research efforts toward adaptive strategy selection.

Prior work has explored various approaches, including mutation operator scheduling [29, 44], runtime strategy switching [46], mutation sequence learning [21], and multi-fuzzer coordination [8, 11]. However, recent work [32, 37] reports that existing fuzzing frameworks suffer from inefficient resource allocation and an inability to adapt to the current state of the fuzzing target. Even well-established algorithms like Thompson Sampling [40] perform no better than random selection in many cases [19, 46]. This observation motivates a key question: if the bottleneck does not lie in the learning algorithm itself, what prevents effective strategy adaptation?

To answer this question, we examine why strategy effectiveness changes during a fuzzing campaign. As the fuzzer discovers new coverage, the program states under exploration evolve because the path constraints that need to be solved change accordingly [4, 42]. Different strategies excel at different types of constraints, so the optimal strategy naturally shifts as the campaign progresses [5]. Therefore, effective learning requires continuously tracking which strategies work under current conditions.

We define the learning cycle as the time interval between observing a strategy's outcome and adapting subsequent choices accordingly. We conduct an empirical study to quantify how fast strategy effectiveness changes. We instrument AFL++'s Havoc stage and measure each mutation operator's contribution to new coverage in one-minute windows. Figure 1 shows the Jaccard distance between top-5 performing operators in adjacent windows, which measures operators set dissimilarity ranging from 0 (identical sets) to 1 (completely disjoint sets). The median distance of 0.75 indicates

Xiaogang Zhu and Sheng Wen are Corresponding authors.

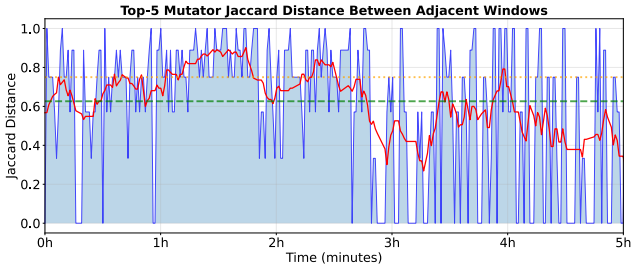


Figure 1: Strategy effectiveness shifts rapidly during fuzzing. We measure Jaccard distance between top-5 performing mutators in adjacent time windows during a 5-hour AFL++ havoc campaign on harfbuzz. Blue lines: 1-minute windows. Red lines: 10-minute windows. Orange dashed line: median distance (0.75). Green dashed line: mean distance (0.6). The high Jaccard distance shows that the best-performing mutation operators change dramatically from one minute to the next, and this drift persists even at 10-minute granularity.

Table 1: Comparison between strategy drift timescale and learning cycles of existing approaches on harfbuzz. Existing methods lag behind strategy drift by 10–191 times. Karamcheti et al.'s TS and MuoFuzz have a fixed learning cycle. The learning cycles of MOpt and EcoFuzz are measured by instrumentation.

Method	**Learning Cycle**
Strategy Drift	∼1 min
Karamcheti et al. [19]	∼10 min
MOpt [29]	∼17 min
MuoFuzz [21]	∼60 min (then fixed)
EcoFuzz [47]	∼191 min (then convergent)

that only 1–2 of the top-5 performing operators in one window remain in the top 5 during the next window. We then compare this drift timescale against the learning cycles of existing approaches. As shown in Table 1, existing methods require 10 to 191 minutes to complete a learning cycle, while strategy effectiveness shifts every minute. This order-of-magnitude mismatch reveals the root cause [3, 41].

Existing fuzzers adopt what we call an in-loop architecture, where strategy selection, mutation, execution, and feedback processing occur sequentially within a single control loop [10]. Under this design, the fuzzer commits to one strategy at a time and must execute it long enough to gather sufficient samples. By the time the fuzzer switches to an alternative, the coverage state has already evolved, making the learning cycle inherently slow regardless of which algorithm is used. The in-loop architecture cannot deliver feedback fast enough to track the rapidly shifting optimal strategy [32, 37]. *Therefore, the bottleneck is not the algorithm, but the current architecture.*

Addressing this architectural limitation is challenging. To observe multiple strategies under similar conditions, multiple execution instances need to execute different strategies in parallel, meaning feedback is distributed. Aggregating this feedback into a coherent global view requires a central component, but communication overhead competes with execution throughput [7, 45]. Furthermore, propagating updated strategy rankings to workers requires coordination that can stall execution. The more frequently the system attempts to adapt, the more overhead it incurs, creating a fundamental tension between adaptation speed and execution efficiency [18, 38].

To address these challenges, we propose TRIOFUZZ, a new fuzzing engine built on a three-tier thread architecture. *Our key insight is that decoupling learning from execution keeps adaptation overhead off the execution hot path, so workers preserve high throughput while the system as a whole adapts within minutes.* Rather than having each worker independently learn and execute, we separate these concerns into specialized thread roles. A single Scheduler thread serves as the centralized adaptation engine, continuously aggregating feedback from all executing threads to maintain a near real-time view of strategy effectiveness. Worker threads focus exclusively on high-speed execution, receiving pre-computed strategy assignments without learning overhead. Scouter threads test underexplored strategies, providing fresh samples to prevent the learner from converging prematurely. When a Scouter discovers an effective strategy, the Scheduler incorporates this signal and potentially promotes the strategy to Workers in subsequent rounds.

We evaluate TRIOFUZZ on benchmarks from FuzzBench [15], OSS-Fuzz [14], and Magma [17]. Our evaluation reveals three key findings. First, TRIOFUZZ consistently outperforms both single-core and multi-core baselines across FuzzBench, OSS-Fuzz, and Magma, demonstrating that the architectural advantage is robust across diverse targets and evaluation criteria. Second, the benefit is most dramatic on complex targets. On sqlite3, TRIOFUZZ triggers 7 vulnerabilities within 30 seconds that baselines require hours or fail to find entirely. Third, our ablation study shows that MOpt, which often underperforms other approaches in recent benchmarks, becomes competitive with them, confirming that the bottleneck has been architectural all along. TRIOFUZZ further discovers 19 new CVEs across 6 real-world projects, all of which have been responsibly disclosed. In summary, our contributions are as follows:

- We propose TRIOFUZZ, a three-tier thread architecture that decouples learning from execution and enables minute-level strategy adaptation across all workers. The same learning algorithms perform significantly better under our proposed architecture.

- We identify the existing in-loop architecture as the key bottleneck preventing effective strategy adaptation, and validate this diagnosis through ablation.

- We conduct extensive evaluations on FuzzBench, OSS-Fuzz, and Magma, showing that TRIOFUZZ outperforms state-of-the-art fuzzers in both coverage and vulnerability discovery, and discovers 19 new CVEs across 6 real-world projects.

2 Background and Related Work

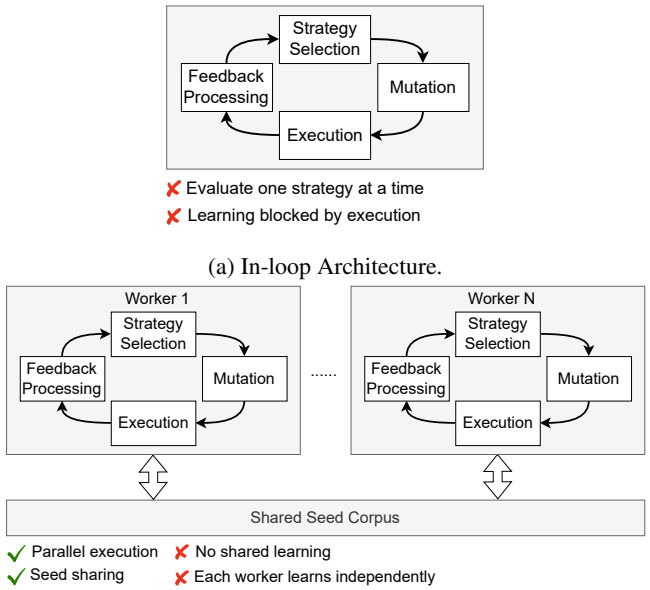
2.1 Coverage-Guided Greybox Fuzzing

Coverage-guided greybox fuzzing (CGF) is a widely adopted technique for automated software testing [30, 51]. A CGF fuzzer maintains a corpus of test inputs and iteratively generates new inputs by mutating existing seeds. The fuzzer executes the target program with each generated input and monitors code coverage through lightweight instrumentation. Inputs that trigger previously unseen coverage are retained in the corpus for further mutation, while others are discarded.

AFL [48] pioneered this approach and inspired numerous extensions. AFLFast [5] models fuzzing as a Markov chain and introduces power schedules to prioritize seeds exercising low-frequency paths. FairFuzz [25] targets rare branches by masking mutations that do not hit underexplored regions. Angora [6] uses gradient descent and taint tracking to solve path constraints efficiently. REDQUEEN [2] leverages input-to-state correspondence to bypass magic byte comparisons without symbolic execution. CollAFL [12] mitigates path collision through more accurate coverage representation. These advances primarily optimize seed scheduling or constraint solving, while TRIOFUZZ focuses on a complementary dimension of adaptive mutation strategy selection.

2.2 Adaptive Strategy Selection

The complementarity of mutation strategies has motivated research on adaptive strategy selection. AFL++'s Havoc [9] applies mutation operators with uniform probability, serving as a non-adaptive baseline. Early work like FuzzSim [43] models blackbox fuzzing as a multi-armed bandit problem, establishing theoretical foundations for adaptive fuzzing. Karamcheti et al. [19] apply Thompson Sampling (TS) to greybox fuzzing but report limited improvements over random selection, a finding we attribute to architectural constraints rather than algorithmic limitations. MOpt [29] uses Particle Swarm Optimization (PSO) to tune mutation operator probabilities, achieving improved coverage over AFL on several benchmarks. MuoFuzz [21] learns conditional probabilities between adjacent operators using a Markov Chain during an initial training phase. Researchers also attempt to introduce Multi-Armed Bandit algorithms to fuzzing. EcoFuzz [47] models fuzzing as an adversarial MAB problem and applies adaptive energy allocation. SLopt [23] proposes a bandit optimization framework that treats mutation operator selection as an online learning problem. Havoc-MAB [44] applies contextual bandits to havoc mutation scheduling.



(a) In-loop Architecture.

(b) Parallel Architecture.

Figure 2: Existing fuzzing architectures. Both approaches limit strategy adaptation: in-loop architectures evaluate one strategy at a time, while parallel architectures lack shared learning across instances.

Despite their algorithmic diversity, all these approaches adopt an in-loop architecture (Figure 2a) where strategy selection, mutation, execution, and feedback processing occur sequentially within a single thread [10]. As our empirical study shows in Table 1, this architecture requires tens of minutes per learning cycle while strategy effectiveness shifts at minute-level timescales, fundamentally limiting adaptation speed regardless of the learning algorithm used.

2.3 Parallel and Collaborative Fuzzing

Parallel fuzzing increases throughput by running multiple fuzzer instances concurrently (Figure 2b). AFL++ parallel mode [9] runs multiple instances sharing a common corpus, with each instance operating independently. EnFuzz [8] coordinates heterogeneous fuzzers through seed synchronization, demonstrating that fuzzer diversity improves coverage. PAFL [27] prioritizes seeds across parallel instances based on coverage potential. autofz [11] dynamically allocates CPU resources among fuzzers based on recent performance. CollabFuzz [34] provides pluggable seed selection policies for collaborative fuzzing. μFuzz [7] redesigns parallel fuzzing using a microservice architecture to reduce synchronization overhead. KRAKEN [49] introduces program-adaptive parallelization to reduce redundant executions. UltraFuzz [50] addresses resource efficiency in distributed fuzzing through coverage-guided task allocation.

Beyond seed synchronization, recent work explores deeper collaboration. Cupid [16] uses send/receive primitives to enable inter-fuzzer communication. Cerebro [26] centralizes coverage tracking to eliminate redundant executions across instances. AFLNET [36] extends parallel fuzzing to stateful network protocols. Nezha [35] coordinates fuzzers for differential testing by sharing semantic information rather than raw seeds. These approaches improve coordination efficiency but still rely on independent per-instance strategy selection.

TRIOFUZZ differs by sharing learned strategy rankings across all execution threads through a centralized Scheduler. Rather than coordinating at the seed level, TRIOFUZZ enables strategy-level collaboration: when one thread discovers an effective mutation strategy, the Scheduler immediately propagates this knowledge to all Workers, enabling system-wide adaptation rather than independent per-instance learning.

3 Design

3.1 Overview

The core insight behind TRIOFUZZ is that effective strategy adaptation requires *decoupling learning from execution*. In existing in-loop architectures, a single thread sequentially selects a strategy, executes test cases, and updates its learning model. This tight coupling bounds the learning cycle to the execution speed of one thread, preventing the fuzzer from observing multiple strategies under comparable conditions.

TRIOFUZZ addresses this limitation through a three-tier thread architecture illustrated in Figure 3. All threads share access to common resources, including the corpus manager, coverage bitmap, and crash storage, while thread-local caches minimize synchronization overhead. At the center, a single Scheduler thread serves as the centralized adaptation engine, operating in a continuous Aggregate-Update-Distribute cycle. The Scheduler *aggregates* feedback from execution threads via Double-Buffered (DB) queues, *updates* strategy rankings through a Pluggable Learning Module, and *distributes* precomputed tasks to execution threads. A hybrid period trigger ensures sub-minute adaptation regardless of target execution speed. At the execution layer, Worker threads and Scouter threads serve complementary roles. Workers focus on intensive execution, pulling complete task specifications from a prefilled queue and executing the same strategy repeatedly to maximize throughput on validated approaches. Scouters focus on strategy diversification, receiving random samples directly from the Learning Module and executing each once to ensure the Scheduler receives signals from all strategies, not just current top performers.

Figure 4 contrasts the coordination mechanisms of existing approaches with TRIOFUZZ. In the in-loop model shown in Figure 4a, an in-loop architecture with a single thread evaluates strategies sequentially, spending tens of minutes on each learning cycle as indicated by the update periods of MOpt

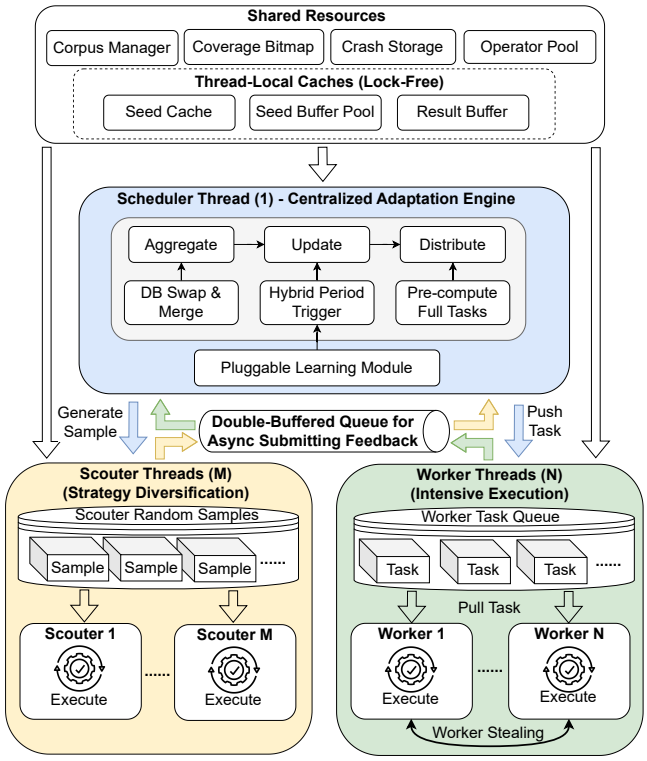
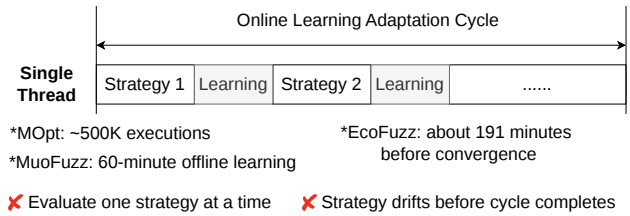
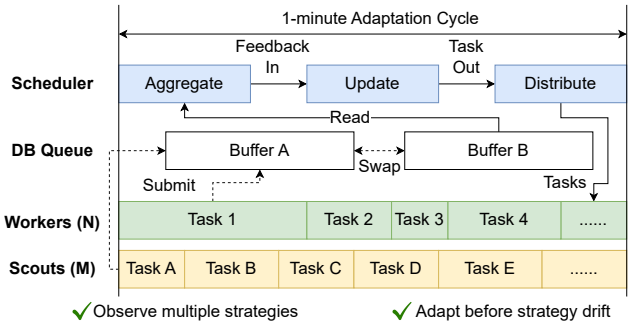


Figure 3: TRIOFUZZ's three-tier thread architecture. The Scheduler thread centralizes all learning decisions through an Aggregate-Update-Distribute cycle, using hybrid period triggers to ensure sub-minute adaptation. A Pluggable Learning Module encapsulates the learning algorithm. Worker threads pull tasks from a prefilled queue and execute repeatedly for throughput. Scouter threads receive random samples directly from the Learning Module, executing each once to prevent premature convergence. Double-buffered queues enable lock-free asynchronous feedback submission.

(500K executions), EcoFuzz (about 191 minutes before convergence), and MuoFuzz (60-minute offline training). By the time the fuzzer finishes evaluating one strategy and begins testing another, the coverage state has already evolved, causing the strategy to drift before the learning cycle completes. In contrast, TRIOFUZZ employs parallel observation as shown in Figure 4b. Multiple Workers and Scouters execute different strategies, submitting results to double-buffered queues. The Scheduler continuously aggregates these results, updates the learning model, and distributes new tasks within a one-minute adaptation cycle. Because the Scheduler observes outcomes from multiple strategies within the same time window, it can compare their effectiveness under similar coverage conditions and adapt before strategy drift occurs.



(a) In-loop Architecture: Sequential Observation



(b) TRIOFUZZ's Architecture: More Strategies Observation

Figure 4: Architectural comparison for learning mechanism between in-loop and TRIOFUZZ's strategy observation. TRIOFUZZ's parallel observation enables adaptation before strategy drift occurs.

3.2 Scheduler Thread

The Scheduler thread is the architectural cornerstone that enables fast adaptation. Unlike in-loop architectures, where learning is distributed across independent threads, TRIOFUZZ centralizes all learning decisions in a single Scheduler. While it eliminates redundant computation by maintaining a single learning model, the Scheduler also enables observing strategy effectiveness across all execution threads simultaneously, providing a global view that no individual thread could achieve alone.

Aggregate-Update-Distribute Cycle. As illustrated in Figure 4b, the Scheduler operates in a three-phase cycle. In the *Aggregate* phase, the Scheduler collects execution feedback from all Workers and Scouters via double-buffered queues. Each queue maintains two buffers that execution threads write to the active buffer while the Scheduler reads from the inactive buffer, swapping atomically to eliminate contention. In the *Update* phase, the Scheduler invokes the Pluggable Learning Module to update strategy rankings based on aggregated feedback. To ensure sub-minute adaptation regardless of target execution speed, TRIOFUZZ employs a hybrid trigger mechanism in which an update occurs when either (1) the total execution count since the last update exceeds a threshold, or (2) wall-clock time since the last update exceeds a period, whichever comes first. In the *Distribute* phase, the Scheduler generates complete task specifications and pushes them to each Worker's task queue.

Algorithm 1: Scheduler Loop with Hybrid Trigger

Input: LearningModule L, ResultQueues R, TaskQueues T, TimeThreshold T_{time}, ExecThreshold T_{exec}

```
1  last_update ← Now(); exec_count ← 0;
2  while running do
       // Aggregate feedback from all
          execution threads
3      foreach queue q in R do
4          results ← q.SwapAndGet();
5          foreach (strategy_id, found_cov) in results
            do
6              L.ProcessResult(strategy_id, found_cov);
7              exec_count ← exec_count + 1;
       // Hybrid trigger: adapt by time OR
          execution count
8      if Now() − last_update > T_time or
        exec_count > T_exec then
9          L.UpdateRankings( );
10         last_update ← Now(); exec_count ← 0;
       // Distribute tasks to Workers
11     foreach queue t in T do
12         while t.size() < target_depth do
13             task ← L.GenerateTask( );
14             t.Push(task);
```

Pluggable Learning Module. The Scheduler interacts with learning algorithms through a minimal interface, enabling different strategies to be integrated without modifying the coordination infrastructure. As shown in Algorithm 1, a learning module implements three operations. `ProcessResult` accepts execution feedback to update internal state, invoked at line 6 for each result aggregated from the double-buffered queues. `UpdateRankings` recomputes strategy rankings based on accumulated feedback, invoked at line 9 when the hybrid trigger fires. `GenerateTask` returns a complete task specification for Workers, invoked at line 13 whenever the Scheduler needs to refill a Worker's task queue. The hybrid trigger at lines 8–10 ensures sub-minute adaptation: an update occurs when either wall-clock time exceeds T_{time} or execution count exceeds T_{exec}, whichever comes first.

3.3 Worker Threads

Worker threads are responsible for high-throughput test execution. Their design aims to minimize the execution path by removing all non-essential computation. Unlike in-loop architectures, where each thread performs strategy selection, mutation, execution, and learning updates, Workers in TRIO-

FUZZ focus exclusively on mutation and execution, with all other responsibilities offloaded to the Scheduler.

Task Queue and Repeated Execution. Each Worker operates in a simple loop, as illustrated in Figure 3. The Worker pulls a task from its private Task Queue, where each task specifies a complete execution plan generated by the Scheduler. Because the Scheduler favors high-ranking strategies when generating tasks, Workers naturally spend more execution cycles on validated approaches without making any selection decisions themselves. The Worker executes each task once, records the coverage outcome, and immediately proceeds to the next task.

Work Stealing. To prevent Workers from stalling, the Scheduler prefills each Worker's private task queue to a configurable depth. When a Worker's queue runs low, the Scheduler's tight loop detects this within microseconds and generates new tasks. If a Worker's queue becomes empty before the Scheduler responds, the Worker attempts to steal tasks from other Workers' queues by taking from the back of a victim's queue while the victim consumes from the front. This combination of prefilled buffering and work stealing ensures Workers remain productive under varying execution speeds.

3.4 Scouter Threads

Scouter threads address a sampling bias challenge inherent to online learning shown in Figure 4. As the Scheduler generates tasks that favor high-ranking strategies, strategies that performed poorly in early iterations receive progressively fewer samples. This creates a feedback loop where the learner lacks sufficient data to detect when a previously ineffective strategy becomes effective under new coverage conditions.

Random Strategy Sampling. Unlike Workers that pull tasks from the Scheduler-managed Task Queue, each Scouter requests random samples directly from the Pluggable Learning Module. The Learning Module selects uniformly across all available strategies, ensuring that every strategy receives periodic evaluation even when the Scheduler concentrates Worker resources on top performers.

Rapid Feedback. Scouters execute each generated sample exactly once and immediately report the outcome to the Learning Module. This tight feedback loop allows the learner to quickly incorporate signals from diverse strategies. When a Scouter discovers new coverage using a low-ranking strategy, this single observation can significantly shift the strategy's ranking, potentially promoting it to the set of strategies that Workers will execute at scale in subsequent rounds.

3.5 Asynchronous Coordination

The three-tier architecture introduces a coordination challenge. Execution threads submit results to the Scheduler, and the Scheduler distributes tasks to Workers, all without introducing synchronization bottlenecks that would negate the throughput benefits of parallel execution. TRIOFUZZ addresses this challenge through three mechanisms illustrated in Figure 3 and Figure 4b.

Double-Buffered Result Queues. Each execution thread submits results to a double-buffered queue. In Figure 4b, each queue maintains two buffers where execution threads write to the active buffer while the Scheduler reads from the processing buffer. The Scheduler periodically swaps the two buffers in a single atomic operation, making the active buffer available for processing while providing a fresh buffer for writers. This design eliminates contention between execution threads and the Scheduler. Execution threads never block waiting for the Scheduler to finish processing, and the Scheduler processes results in batches rather than one at a time, amortizing the cost of strategy updates across many results.

Shared resources. Figure 3 shows the shared resources for all threads. The corpus manager maintains the seed corpus that allows concurrent reads during seed selection. The coverage bitmap uses atomic compare-and-swap operations to record new coverage, ensuring that coverage checks remain fast even under high contention. The crash storage handles concurrent crash submissions. The operator pool provides access to mutation operator definitions, requiring no synchronization.

Thread-local caches. As shown in Figure 3, each execution thread maintains thread-local caches to reduce access to shared resources. A seed cache holds recently fetched seeds, allowing the thread to perform multiple mutations without repeatedly acquiring locks on the global corpus manager. A seed buffer pool provides pre-allocated memory for mutation operations, avoiding repeated allocation overhead. A result buffer accumulates execution outcomes locally before submitting them in batches to the double-buffered queue. These caches convert frequent fine-grained accesses into infrequent coarse-grained accesses, reducing lock contention proportionally to the cache size.

3.6 Default Learning Implementation

The Pluggable Learning Module accepts learning algorithms through the interface defined in Algorithm 1. To isolate the architectural contribution from algorithmic factors, our default implementation deliberately reuses well-established algorithms from prior work, including Thompson Sampling [19], Particle Swarm Optimization [29], Markov Chain [21], and adaptive energy allocation [47], rather than proposing new ones. This design choice enables a controlled experiment.

Our default learning instantiation organizes these algorithms in a two-layer hierarchy, as detailed in Algorithm 2. The outer layer addresses *strategy arm selection*. Each strategy arm specifies a mutation mode (Havoc or MOpt), whether to use EcoFuzz's adaptive energy allocation [47] for seed scheduling, and whether to enable splice. These strategy arms compete on equal footing through Thompson Sampling. An `Analysis` arm that invokes heavyweight techniques such as

Algorithm 2: Two-Layer Strategy Learning

Input: StrategyArms $\{\text{arm}_1, \ldots, \text{arm}_K\}$, TS parameters $\{\alpha_i, \beta_i\}$

1 **Function** `GenerateTask()`
    `// Outer layer: TS selects strategy arm`
2 $arm \leftarrow \arg\max_i \text{Sample}(\text{Beta}(\alpha_i, \beta_i))$;
3 $(use_mopt, splice) \leftarrow$ `Config`$[arm]$;
4 $stack \leftarrow 1 \ll (1 + \text{Rand}(\texttt{StackPow2}))$;
    `// Inner layer: operator weights + sequencing`
5 **if** *use_mopt* **then** $weights \leftarrow$ `PsoSample()`;
6 **else** $seq \leftarrow$ `MarkovSample`$(stack, weights)$;
7 **return** `Task`$(arm, splice, seq)$;
8 **Function** `ProcessResult` (*task, found_cov*)
    `// Update outer layer`
9 **if** *found_cov* **then** $\alpha[task.arm] \mathrel{+}= 1$;
10 **else** $\beta[task.arm] \mathrel{+}= 1$;
    `// Update inner layer`
11 `Pso.Update`$(task.seq, found_cov)$;
12 `Markov.Update`$(task.seq, found_cov)$;

comparison logging (CmpLog) is additionally available but restricted to Scouter threads, as its per-execution overhead is incompatible with Worker throughput goals. At line 2, TS samples from Beta distributions maintained for each strategy arm and selects the one with the highest sampled value. The inner layer addresses *strategy execution*. Once a strategy arm is selected, two learning algorithms jointly refine how it generates mutations. When the selected arm specifies MOpt mode, PSO adjusts the probability distribution over mutation operators (line 5); in Havoc mode, MuoFuzz's Markov Chain samples a sequence of operators by learning transition probabilities between adjacent operators (line 6), capturing which operator combinations are effective.

In `ProcessResult`, the outer layer updates the Beta distribution for the selected strategy arm at lines 9–10. Successful coverage discovery increments α, increasing future selection probability, while failure increments β. The inner layer updates at lines 11–12, where PSO adjusts particle positions based on coverage feedback, and the Markov Chain updates transition probabilities for the executed operator sequence.

4 Evaluation

4.1 Experimental Setup

Implementation. We implement TRIOFUZZ from scratch in approximately 35K lines of C++ code. The implementation includes the three-tier thread architecture, the unified learning module as the default described in Section 3.

Baselines. We compare TRIOFUZZ against six fuzzers on FuzzBench [15]: AFL++ [9], AFL++ parallel [9], LibFuzzer [28], MOpt [29], MuoFuzz [21], and EcoFuzz [47]. On OSS-Fuzz [14], we compare against AFL++, MOpt, MuoFuzz, and EcoFuzz. Honggfuzz is excluded from the FuzzBench and OSS-Fuzz experiments due to build failures on several targets, but is included in the Magma evaluation, where all targets compile successfully. For AFL++, LibFuzzer, MOpt, and EcoFuzz, we use the versions integrated in FuzzBench. For MuoFuzz, we use the latest version available on GitHub.

Benchmarks. We evaluate on three benchmark suites. For coverage evaluation, we use 9 targets from FuzzBench and 10 projects from OSS-Fuzz. For vulnerability detection, we use Magma and additionally fuzz 6 real-world open-source projects. We select OSS-Fuzz projects based on two criteria. First, we ensure diversity in code complexity, with branch counts ranging from 50 to 85K. Second, we cover a variety of application domains, including parsers (e.g., rapidjson, yaml-cpp), image and multimedia processing (e.g., libaom, libexif), compression (e.g., miniz), networking (e.g., c-ares), and system utilities (e.g., cmake, libunwind, capstone, cppitertools). We select targets that are thread-safe, as TRIOFUZZ's multi-threaded architecture requires concurrent execution against the same target binary. We exclude targets such as libxml2 [1] and freetype2 [39] that are documented as thread-unsafe or exhibit non-deterministic crashes under parallel execution.

Configuration and Environment. All experiments run on a machine with AMD Ryzen Threadripper PRO 5975WX (32 cores, 64 threads) and 128GB RAM in Ubuntu 22.04. TRIOFUZZ runs with 4 CPU cores allocated as 1 Scheduler, 2 Workers, and 1 Scouter across all benchmarks. For FuzzBench experiments (Section 4.2), all fuzzers run for 24 hours with 10 trials. We include AFL++ under two configurations: single-core mode and 4-core master-slave parallel mode, enabling comparison against both sequential and parallel baselines. Other baseline fuzzers, including LibFuzzer, MOpt, MuoFuzz, and EcoFuzz, run on a single CPU core following their standard configurations. For OSS-Fuzz and Magma experiments (Sections 4.3 and 4.4), we compare under an equal CPU-hours budget: TRIOFUZZ runs for 6 hours with 4 cores (24 CPU-hours), while all baselines run for 24 hours on a single core (24 CPU-hours). This configuration evaluates whether TRIOFUZZ can achieve comparable or superior results within the same total computational budget but significantly less wall-clock time. In the ablation study (Section 4.7), we evaluate TRIOFUZZ under different thread configurations to analyze the impact of Worker-to-Scouter allocation.

Our evaluation aims to answer the following questions:

- **RQ1:** Does TRIOFUZZ achieve higher code coverage than state-of-the-art fuzzers?
- **RQ2:** Does TRIOFUZZ find more vulnerabilities?
- **RQ3:** Is the performance improvement attributable to the architecture rather than the learning algorithm?

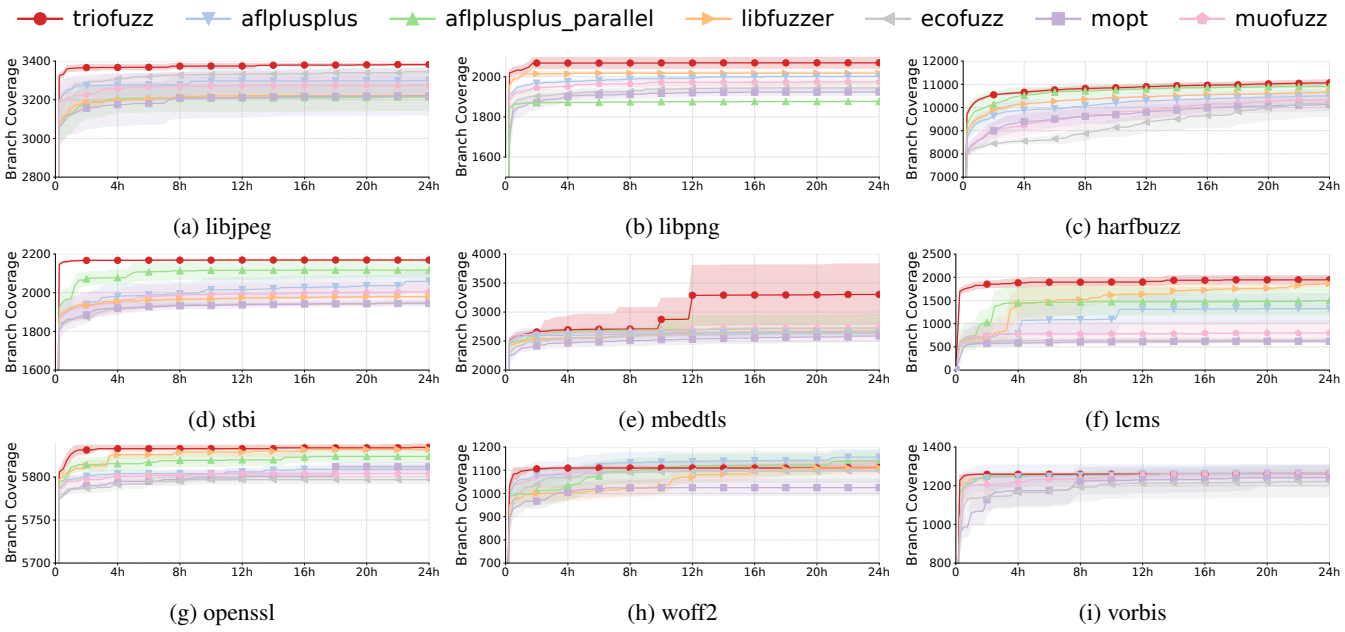


Figure 5: Branch coverage over 24 hours on 9 FuzzBench targets with 10 trials. Our TRIOFUZZ performs best on 7 out of 9 targets.

4.2 RQ1: Coverage Comparison on Fuzzbench

Figure 5 shows the branch coverage over 24 hours for 9 FuzzBench targets with 10 trials. TRIOFUZZ achieves the highest final coverage on 7 out of 9 targets and remains competitive on the remaining two. The consistent improvement over all learning-based baselines demonstrates that decoupling learning from execution enables more effective strategy adaptation than optimizing the learning algorithm alone.

On woff2 and vorbis, TRIOFUZZ matches the best baselines in mean coverage, although individual trials of certain baselines occasionally reach slightly higher peak values. We attribute this to the relatively constrained input structure of these targets, where the strategy space is narrow enough that even non-adaptive approaches can reach near-optimal coverage. Nevertheless, TRIOFUZZ does not exhibit performance degradation on these targets, indicating that the three-tier architecture introduces no penalty even when rapid strategy adaptation is less beneficial.

The comparison between TRIOFUZZ and AFL++ parallel is informative, as both configurations utilize 4 CPU cores yet exhibit consistent performance differences across most targets. This gap cannot be attributed to computational resources, but rather to how parallel resources are coordinated. As shown in Figure 2b, AFL++ parallel runs independent instances that share a corpus but make strategy decisions in isolation. When one instance discovers an effective mutation strategy, this knowledge remains local. In contrast, TRIOFUZZ's Scheduler aggregates feedback from all execution threads and propagates updated strategy rankings system-wide within minutes. Notably, AFL++ parallel achieves only marginal improvement over its single-core configuration on most targets despite consuming 4 times the resources, confirming that parallelism alone, without shared learning, does not enable effective strategy-level collaboration.

Listing 1: TRIOFUZZ Covers a Unique Path in Libpng Requiring Multiple Magic Byte Constraints.

```
1  /* Constraint 1: Valid PNG header */
2  /* Input must begin with 8-byte magic signature:
       */
3  /* 89 50 4E 47 0D 0A 1A 0A */
4  /* Constraint 2: Correct chunk type */
5  if (png_memcmp(chunk_name, "iCCP", 4) == 0) {
6      /* Constraint 3: Valid ICC profile magic */
7      if (profile_length >= 4 &&
8          png_memcmp(profile, "acsp", 4) == 0) {
9          /* Constraint 4: CRC32 checksum must match
                */
10         if (crc_check(chunk_data, length) == 0) {
11             /* Rare ICC profile processing path */
12             png_icc_check_header(png_ptr, ...);
13         }
14     }
15 }
```

To illustrate why TRIOFUZZ achieves higher coverage than baselines, we analyze a representative rare path in `libpng` that TRIOFUZZ covers only. Listing 1 shows the iCCP chunk processing path in `pngrutil.c`. Triggering this path demands four conditions. At lines 1–3, the input needs to carry a valid

Table 2: Line and Branch (Br.) coverage results on OSS-Fuzz benchmarks (24 CPU-hours, mean ± std with 10 trials).

Project	#Br.	TrioFuzz		AFL++		MOpt		MuoFuzz		EcoFuzz	
		Line	Br.	Line	Br.	Line	Br.	Line	Br.	Line	Br.
capstone	85130	**218442**±313	**42840**±330	216028±512	39378±315	215260±370	38943±312	216164±450	39490±153	212571±743	36088±198
libaom	21171	**46410**±469	**12688**±132	45452±286	12180±82	44480±140	11700±81	45587±292	12204±86	43582±126	11161±97
c-ares	8721	3759±3	1975±2	**3790**±3	**2002**±2	3741±12	1967±6	3787±3	2001±2	3716±34	1956±15
cmake	7831	**3543**±12	**2688**±5	3128±74	2448±53	3160±69	2437±59	3168±54	2435±44	3242±37	2471±33
libexif	3834	4051±20	2650±11	**4081**±1	**2677**±5	4014±20	2612±21	3097±29	2520±35	3992±32	2598±22
miniz	3186	**1337**±34	**742**±24	1274±13	693±7	1280±31	699±15	1185±11	618±6	1176±6	626±11
libunwind	2448	**2041**±1	**698**±2	1968±0	646±0	1968±0	647±0	1968±0	647±0	1968±0	646±0
yaml-cpp	1180	**2162**±0	**972**±1	**2162**±0	**972**±1	2155±5	963±5	2148±2	959±2	2158±2	965±1
rapidjson	2947	2119±8	**966**±5	2116±8	961±3	2091±13	938±9	**2120**±4	963±5	2083±17	935±11
cppitertools	86	**403**±1	**74**±1	402±0	73±0	402±0	73±0	402±0	73±0	402±0	73±0
Avg	13653	**28427**	**6629**	28040	6203	27855	6098	27963	6191	27489	5752

PNG header with the 8-byte magic signature `89 50 4E 47 0D 0A 1A 0A`; any corruption causes the parser to reject the file before chunk processing begins. At line 5, the chunk type field needs to match the 4-byte string `"iCCP"`, which identifies the embedded ICC color profile chunk. At lines 7–8, the ICC profile payload needs to begin with the `"acsp"` magic bytes that mark a valid ICC profile header. Finally, at lines 9–10, a CRC32 checksum computed over the chunk data must match the value stored in the chunk trailer. Under in-loop architectures, strategy rankings update too slowly to track this shifting bottleneck. By the time the fuzzer recognizes that the magic byte constraint has been resolved, the coverage state has already evolved past the point where this feedback is actionable. TrioFuzz's sub-minute adaptation enables the Scheduler to detect which strategy is currently productive, rapidly shifting resources from dictionary-based mutations to arithmetic mutations as constraints are resolved one by one.

4.3 RQ1: Coverage Comparison on OSS-Fuzz

Table 2 shows the line and branch coverage results. TrioFuzz achieves the highest or tied-highest branch coverage on 8 out of 10 projects, despite running for only one-quarter of the wall-clock time. On average across all 10 projects, TrioFuzz reaches 6,629 branches compared to 6,203 for AFL++, 6,098 for MOpt, 6,191 for MuoFuzz, and 5,752 for EcoFuzz.

The advantage is pronounced on complex targets with diverse code paths. On libaom, the multimedia target with 21,171 branches, TrioFuzz reaches 12,688 branches compared to 12,180 for AFL++ and 11,161 for EcoFuzz. On cmake, TrioFuzz reaches 2,688 branches versus 2,448 for AFL++ and 2,471 for EcoFuzz, and also attains the highest line coverage at 3,543. These targets expose many distinct paths that benefit from rapid strategy switching between structural and byte-level mutations. On capstone, the largest target with 85,130 branches, TrioFuzz reaches 42,840 branches versus 39,490 for MuoFuzz and 39,378 for AFL++. TrioFuzz also attains the highest line coverage on this target at 218,442, ahead of 216,164 for MuoFuzz and 216,028 for AFL++. The lead on both line and branch coverage indicates that the architectural advantage scales to the most complex target in our suite.

Table 3: Average number of vulnerabilities triggered on Magma in 24 CPU-hours with 10 trials

Target	Trio	A++	HF	LF	MO	MF	EF
libpng	**4.0**	2.8	3.4	2.2	1.6	2.8	1.0
libsndfile	**7.0**	**7.0**	6.6	6.0	**7.0**	**7.0**	**7.0**
libtiff	**4.4**	3.6	3.0	1.6	4.0	3.8	3.0
openssl	2.0	**3.0**	2.8	1.0	2.0	2.0	**3.0**
php	**3.0**	**3.0**	**3.0**	1.0	1.0	**3.0**	2.0
poppler	**3.0**	2.8	2.8	2.2	2.0	**3.0**	2.5
sqlite3	**7.0**	3.8	3.6	1.2	3.6	4.4	1.5
Total	**30.4**	26.0	25.2	15.2	21.2	26.0	20.0

Trio=TrioFuzz, *A++=AFL++, HF=Honggfuzz, LF=LibFuzzer, MO=MOpt, MF=MuoFuzz, EF=EcoFuzz*

On c-ares and libexif, TrioFuzz does not reach the top branch coverage. On c-ares, AFL++ reaches 2,002 branches, and MuoFuzz reaches 2,001, while TrioFuzz reaches 1,975. On libexif, AFL++ reaches 2,677 while TrioFuzz reaches 2,650. Both targets are small with well-defined input formats and limited strategy diversity. A 24-hour single-core campaign benefits from sustained sequential exploration on such targets. The narrow strategy space offers fewer opportunities for adaptive switching to improve over a well-tuned default configuration. TrioFuzz still surpasses MOpt and EcoFuzz on c-ares, and surpasses all baselines except AFL++ on libexif.

4.4 RQ2: Vulnerabilities Found on Magma

We evaluate TrioFuzz's vulnerability-finding capability on the Magma benchmark [17]. Table 3 shows the average number of unique vulnerabilities triggered by each fuzzer over 10

Table 4: Time to trigger vulnerabilities on Magma benchmark (24 CPU-hours, 10 trials). TRIOFUZZ achieves the fastest discovery on 22 of 37 vulnerabilities.

Target	Bug	Trio	A++	HF	LF	MO	MF	EF	Target	Bug	Trio	A++	HF	LF	MO	MF	EF
libpng	PNG001	**4m**	-	1h	-	-	5d	4d	php	PHP004	**45s**	10h	-	-	9m	1m	5d
	PNG003	30s	19s	39m	**15s**	**15s**	**15s**	48s		PHP009	**45s**	8h	-	-	22m	2m	5d
	PNG006	45s	1m	26m	**22s**	-	1d	1d		PHP011	45s	24m	**15s**	**15s**	27s	1m	2m
	PNG007	**2h**	1d	4d	5d	2d	1d	6m	poppler	PDF010	5h	2h	**45m**	1h	10h	2h	1h
libtiff	TIF002	5d	4d	-	-	**1d**	4d	5d		PDF011	-	**4d**	-	-	-	-	-
	TIF007	**15s**	1m	**15s**	25m	27s	58s	3m		PDF014	-	-	**5d**	-	-	-	-
	TIF008	**5d**	**5d**	-	-	**5d**	**5d**	-		PDF016	2m	**1m**	**1m**	5m	2m	**1m**	4m
	TIF012	**2m**	32m	12m	5d	4m	3m	2h		PDF018	**5h**	4d	5d	5d	-	-	5d
	TIF014	**1m**	1h	4m	4d	14m	3m	2h		PDF019	-	-	-	-	-	**5h**	-
libsndfile	SND001	**1m**	33m	4m	15h	8m	11m	2m		PDF021	-	-	**4d**	-	-	-	-
	SND005	13m	59m	30m	10m	**1m**	2h	4m	sqlite3	SQL002	**30s**	1h	25m	9h	46m	15m	5h
	SND006	**4m**	1h	2d	21h	10m	39m	4h		SQL003	**30s**	-	-	-	-	**4d**	-
	SND007	**2m**	59m	10m	21h	9m	1h	**2m**		SQL012	**30s**	5d	3d	-	-	5d	-
	SND017	**30s**	12m	1h	38m	1m	17m	3h		SQL013	**30s**	-	-	-	5d	-	-
	SND020	**45s**	19m	1h	-	8m	56m	2h		SQL014	**30s**	10h	8h	-	8h	1d	1d
	SND024	7m	53m	3m	21h	9m	34m	**2m**		SQL015	-	-	-	-	-	**4d**	-
openssl	SSL001	-	6h	1d	-	-	-	18h		SQL018	**15s**	1h	1h	5d	8h	26m	5d
	SSL002	**1m**	2m	4m	-	**1m**	**1m**	3m		SQL020	**30s**	3d	-	-	4d	2d	-
	SSL003	15s	3m	**10s**	32s	1m	**10s**	2m	**Fastest**	**Count**	**22**	3	7	3	4	8	2

Trio=TRIOFUZZ, *A++=AFL++, HF=Honggfuzz, LF=LibFuzzer, MO=MOpt, MF=MuoFuzz, EF=EcoFuzz*

Table 5: New vulnerabilities discovered by TRIOFUZZ

Project	CVE ID	Type	CWE
Fio	CVE-2025-10823	NPD	CWE-476
	CVE-2025-10824	UAF	CWE-416
Open Babel	CVE-2025-10994	UAF	CWE-416
	CVE-2025-10995	MC	CWE-119
	CVE-2025-10996	HBO	CWE-122
	CVE-2025-10997	HBO	CWE-122
	CVE-2025-10998	NPD	CWE-476
	CVE-2025-10999	NPD	CWE-476
	CVE-2025-11000	NPD	CWE-476
Libucl	CVE-2025-11010	HBO	CWE-122
BehaviorTree	CVE-2025-11011	NPD	CWE-476
	CVE-2025-11012	SBO	CWE-121
	CVE-2025-11013	NPD	CWE-476
Ogre	CVE-2025-11014	HBO	CWE-122
	CVE-2025-11015	MC	CWE-119
	CVE-2025-11017	NPD	CWE-476
Assimp	CVE-2025-11274	AoR	CWE-770
	CVE-2025-11275	HBO	CWE-122
	CVE-2025-11277	HBO	CWE-122

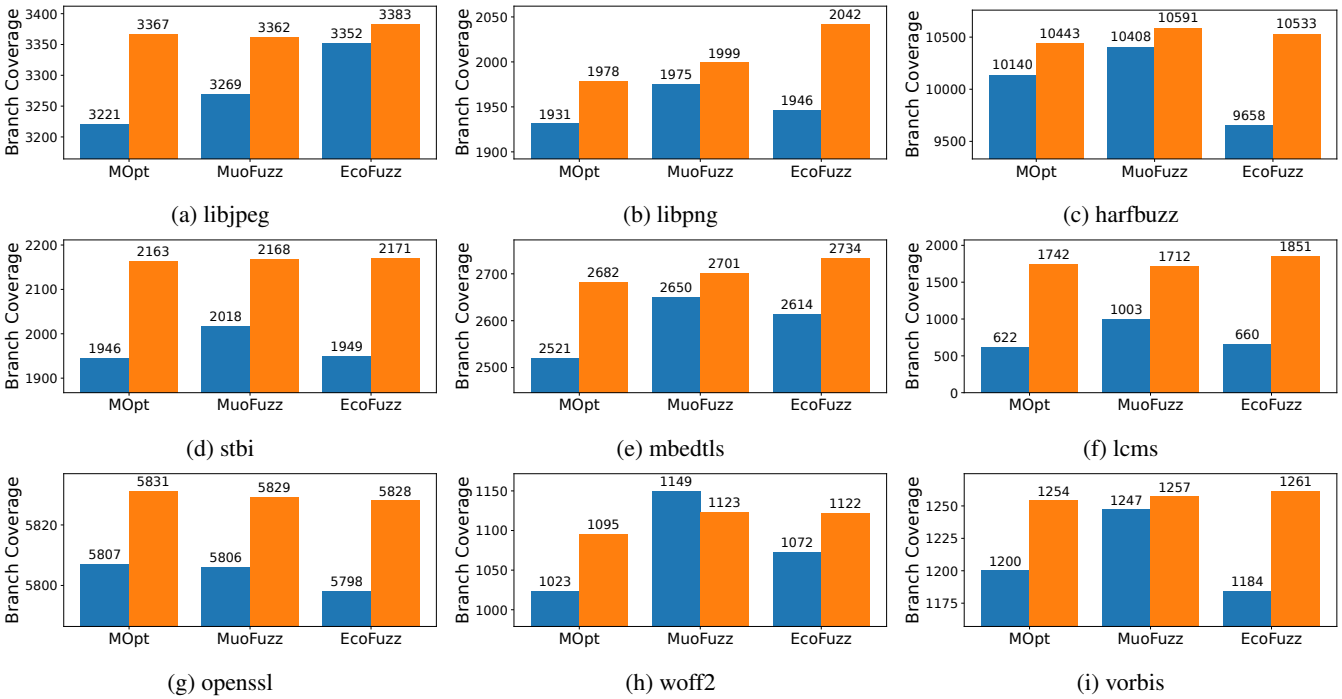
NPD: NULL Pointer Dereference, UAF: Use After Free, HBO: Heap Buffer Overflow, SBO: Stack Buffer Overflow, MC: Memory Corruption, AoR: Allocation of Resources.

trials within 24 CPU-hours. TRIOFUZZ triggers 30.4 vulnerabilities on average, outperforming all baselines: AFL++ and MuoFuzz each trigger 26.0, Honggfuzz triggers 25.2, MOpt triggers 21.2, EcoFuzz triggers 20.0, and LibFuzzer triggers 15.2. Compared to AFL++ and MuoFuzz, TRIOFUZZ discovers 17% more vulnerabilities.

The improvement is most pronounced on sqlite3, where TRIOFUZZ triggers 7 vulnerabilities compared to 3.8 for AFL++ and 4.4 for MuoFuzz. Specifically, TRIOFUZZ triggers sqlite3's 7 vulnerabilities within 30 seconds, whereas baselines either miss them entirely or require hours to days. SQLite3's complex query processing engine presents diverse path constraints that shift rapidly as the fuzzer explores different SQL parsing and execution paths, making it an ideal target for TRIOFUZZ's fast strategy adaptation. On libpng, TRIOFUZZ triggers 4.0 vulnerabilities compared to 2.8 for AFL++, benefiting from the same multi-constraint path coverage advantage demonstrated in our FuzzBench case study (Listing 1).

One exception appears on openssl, where TRIOFUZZ triggers 2 vulnerabilities compared to 3 for AFL++. This is because TRIOFUZZ employs an in-process fuzzing model for higher throughput, whereas AFL++ uses a fork server that can terminate long-running executions via timeout. The version of openssl in Magma often executes for extended periods due to mutated inputs. TRIOFUZZ cannot kill these executions without terminating the entire process. As a result, all of TRIOFUZZ's threads stall for a long time and stop making progress.

Beyond total vulnerability counts, Table 4 reports the time



(a) libjpeg (b) libpng (c) harfbuzz

(d) stbi (e) mbedtls (f) lcms

(g) openssl (h) woff2 (i) vorbis

Figure 6: Comparison of learning algorithms under their native in-loop architecture (blue) versus TRIOFUZZ's three-tier architecture (orange) on 9 FuzzBench targets (24 CPU-hours, 10 trials) with mean code branch coverage. The same algorithms consistently achieve higher coverage under TRIOFUZZ's decoupled architecture.

to first trigger for each vulnerability. TRIOFUZZ achieves the fastest discovery on 22 out of 37 vulnerabilities, compared to MuoFuzz (8), Honggfuzz (7), MOpt (4), AFL++ and LibFuzzer (3 each), and EcoFuzz (2). This speed advantage is the primary driver behind the higher total bug counts in Table 3: within a fixed 24 CPU-hours budget, faster discovery translates directly to more vulnerabilities found. These results demonstrate that TRIOFUZZ's sub-minute strategy adaptation is particularly effective for reaching deep program states quickly, as the Scheduler can rapidly identify and amplify whichever mutation strategy is currently making progress toward triggering a vulnerability.

4.5 RQ2: Identified New Vulnerabilities

To evaluate TRIOFUZZ on previously unknown vulnerabilities, we fuzz 6 popular open-source projects: Fio, Open Babel, Libucl, BehaviorTree.CPP, Ogre, and Assimp. Table 5 lists the 19 new CVEs discovered by TRIOFUZZ. The vulnerabilities span diverse types, including 7 NULL pointer dereferences (CWE-476), 6 heap buffer overflows (CWE-122), 2 use-after-free bugs (CWE-416), 2 memory corruption issues (CWE-119), 1 stack buffer overflow (CWE-121), and 1 allocation of resources without limits (CWE-770). Open Babel contains the most vulnerabilities (7 CVEs), followed by Ogre and BehaviorTree.CPP (3 CVEs each). All vulnerabilities have been reported to the maintainers and assigned CVE identifiers.

4.6 RQ3: Architectural Contribution

To isolate the architectural contribution from algorithmic factors, we evaluate whether the same learning algorithms perform better under TRIOFUZZ's decoupled architecture than under their original in-loop implementations. We extract the core learning algorithms from MOpt, MuoFuzz, and EcoFuzz and integrate them into TRIOFUZZ's Pluggable Learning Module described in Section 3.6, preserving the original algorithm logic while replacing the in-loop feedback mechanism with TRIOFUZZ's Scheduler. As shown in Figure 6, we compare each algorithm under two configurations: its native in-loop implementation running on a single core for 24 hours (blue), and TRIOFUZZ with the same algorithm running on 4 cores for 6 hours (orange). Both configurations consume 24 CPU-hours total and run 10 trials on the 9 FuzzBench targets.

Figure 6 demonstrates that all three algorithms achieve higher coverage under TRIOFUZZ's architecture on nearly every target. The improvement is consistent regardless of which learning algorithm is used, confirming that the performance gain stems from the decoupled architecture rather than from any particular algorithmic advantage. Across the 27 algorithm-target combinations (3 algorithms $\times$ 9 targets), 26 show improvement under TRIOFUZZ's architecture, with only one marginal exception for MuoFuzz in woff2.

A notable finding concerns MOpt. Under its native in-loop architecture, MOpt ranks last on the majority of targets. How-

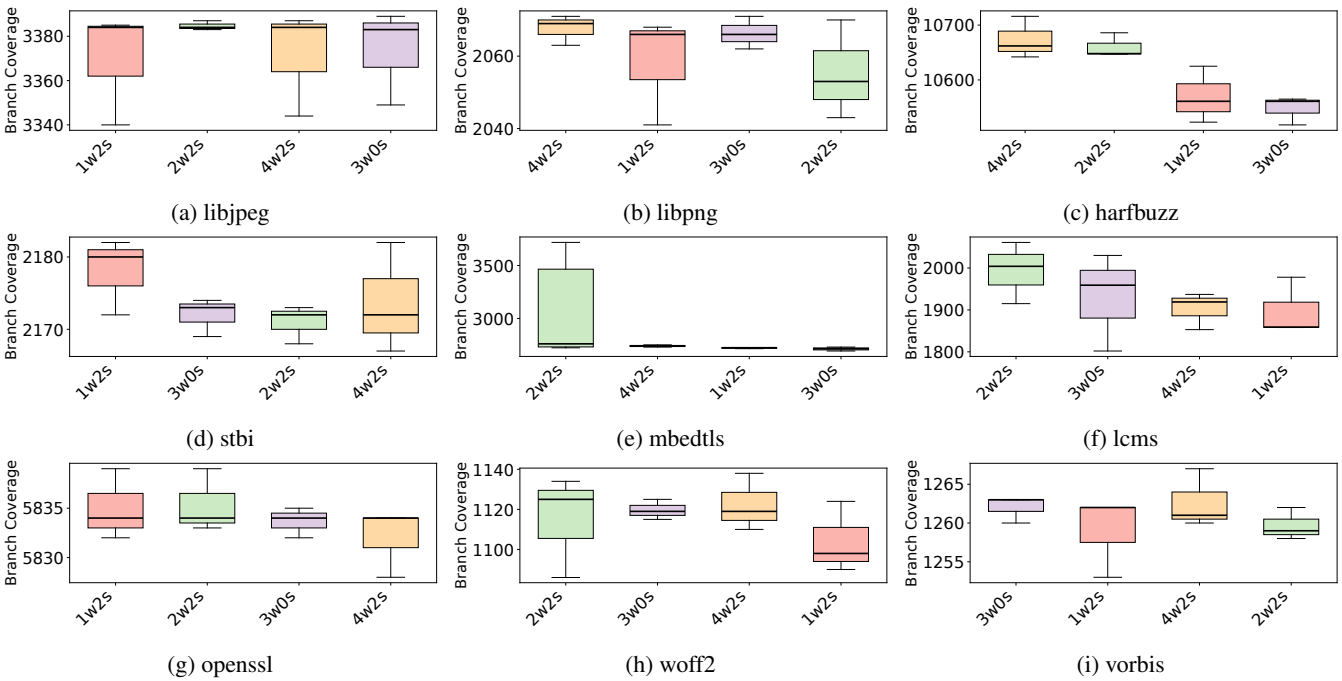


Figure 7: Ablation study of TRIOFUZZ over 1 hour on 9 FuzzBench targets with 10 trials. No single configuration (W: Workers, S: Scouters) can perform best across all targets. *xWyS* denotes *x* Worker threads and *y* Scouter threads.

Table 6: Ablation study on thread configurations (W: Workers, S: Scouters). Each thread uses a single CPU core. Results show average executions per second within 1 hour.

Target	1W2S	2W1S	2W2S	3W0S	4W2S
libpng	20106	18811	24408	18539	**29215**
lcms	8917	5159	**10167**	7235	9559
stbi	2340	1298	1696	**3926**	2625
vorbis	2830	5968	8850	2736	**13796**
openssl	17478	19306	18373	16951	**22956**
mbedtls	10754	8767	15897	12766	**16991**
woff2	28345	30145	**30443**	25742	28023
harfbuzz	2929	2454	5160	3330	**7186**
libjpeg	2947	2558	3868	3026	**4640**
Mean	10738	10496	13207	10472	**14999**

ever, under TRIOFUZZ's decoupled architecture, MOpt exhibits one of the largest relative improvements and becomes competitive with or surpasses the other algorithms. The transformation is dramatic on lcms, where MOpt's branch coverage increases from 622 to 1,742, a nearly 3 times improvement. Similar gains appear on mbedtls (2,521 to 2,682), libjpeg (3,221 to 3,367), and vorbis (1,200 to 1,254). This reversal demonstrates that architectural constraints are a significant contributing factor to MOpt's underperformance. MOpt's PSO continuously adjusts mutation operator weights, making it sensitive to feedback latency. Under in-loop architectures, the slow learning cycle causes PSO particles to converge on outdated program state, whereas TRIOFUZZ's sub-minute feedback enables PSO to track the shifting landscape effectively.

MuoFuzz and EcoFuzz also benefit from the decoupled architecture. EcoFuzz shows improvement across all 9 targets, with the largest gain on lcms (660 to 1,851). MuoFuzz improves on 8 of 9 targets, with notable gains on lcms (1,003 to 1,712) and stbi (2,018 to 2,168). The one exception is woff2, where MuoFuzz shows a marginal decrease from 1,149 to 1,123 branches.

4.7 Ablation Study

To understand how the allocation of Worker and Scouter threads affects performance, we conduct an ablation study with five thread configurations on the 9 FuzzBench targets. Table 6 reports executions per second (throughput), and Figure 7 shows the branch coverage distributions over 1 hour. We focus on the first hour because the majority of coverage growth occurs during this early phase, making it the most informative window for evaluating architectural efficiency.

Table 6 shows that mean throughput scales with the number of execution threads, where *xWyS* denotes *x* Worker threads and *y* Scouter threads. 4W2S, which dedicates six cores to execution, attains the highest mean throughput (14,999 exec/s), followed by 2W2S (13,207 exec/s) with four execution

threads. The three configurations with three execution threads, 1W2S, 2W1S, and 3W0S, cluster together between 10,472 and 10,738 exec/s. This ordering is expected, since aggregate throughput grows with the number of concurrently executing threads regardless of their roles. However, Figure 7 reveals that higher throughput does not consistently translate into higher coverage. The configuration with the highest throughput does not always achieve the best coverage, demonstrating that *what* the fuzzer executes matters as much as *how fast* it executes.

The optimal ratio between Workers and Scouters depends on the characteristics of the target program. On structurally simple targets such as libpng, the all-Worker configuration 3W0S achieves the highest coverage in Figure 7b, even though it does not attain the highest throughput on this target. When the strategy space is narrow, the productive strategy stays stable, so dedicating all execution cores to Workers that exploit a small set of validated strategies pays off more than spending cores on Scouters. The exploration that Scouters provide is largely redundant here, as few alternative strategies offer meaningful gains. Conversely, on complex targets with diverse and rapidly shifting code paths, Scouter threads become essential. On harfbuzz, 4W2S achieves the highest coverage in Figure 7c. Without Scouters, all Workers commit to the same Scheduler-selected strategy, and when that strategy happens to produce inputs that trigger expensive code paths, the Workers stall at sub-optima. The Scheduler, receiving only homogeneous feedback from a single strategy, lacks the signal diversity to recognize that alternative strategies could explore new coverage more effectively. By continuously sampling diverse strategies, Scouters supply the Scheduler with the signals needed to redirect Workers toward more productive strategies. This demonstrates that Scouter-driven strategy diversity translates into tangible coverage gains on complex targets, where no single fixed strategy remains optimal throughout the campaign.

5 Discussion

Our ablation study demonstrates that the same learning algorithms perform differently depending on the underlying architecture. This suggests that future research on adaptive fuzzing should treat architectural design as a primary concern rather than assuming the execution infrastructure is fixed. We believe this insight generalizes beyond fuzzing: any online learning system where the observation cycle is slower than the environment's rate of change will suffer similar degradation.

Our ablation study challenges the common assumption that higher throughput leads to better coverage. The configuration with the highest mean throughput (4W2S) does not achieve the best coverage on every target, while on libpng, the all-Worker 3W0S configuration attains the highest coverage despite not having the highest throughput on that target. The benefit of Scouter threads is target-dependent. On structurally simple targets, a narrow strategy space makes exploration largely redundant, whereas on complex targets such as harfbuzz, where the productive strategy shifts frequently, Scouter-driven strategy diversity is essential for the Scheduler to redirect Workers toward more effective strategies. This suggests that raw execution speed and strategy diversity are distinct objectives, and future parallel fuzzing designs should balance them according to target characteristics rather than maximizing throughput alone.

Extending TRIOFUZZ to stateful network protocols is a future direction, and the ProFuzzBench benchmark [33] offers a suitable testbed for evaluating per-state strategy adaptation.

Threats to validity. Our equal-budget comparison allocates 4 cores $\times$ 6 hours for TRIOFUZZ versus 1 core $\times$ 24 hours for baselines. While both consume 24 CPU-hours, the shorter wall-clock time gives TRIOFUZZ less opportunity for late-stage coverage growth. We use branch coverage as the primary metric following standard practice, though it does not capture all dimensions of testing effectiveness. Finally, as we discover that harfbuzz's branch coverage continuously increases during fuzzing, our strategy drift measurement (Figure 1) is conducted on harfbuzz; while we expect similar patterns on other targets, the exact timescale may vary.

Limitations. TRIOFUZZ requires thread-safe targets, excluding programs such as libxml2 [1] and freetype2 [39] that are documented as thread-unsafe. Its in-process fuzzing model cannot terminate long-running executions without killing the entire process, as observed on openssl in Section 4.4. The optimal Worker-to-Scouter ratio is target-dependent (Section 4.7), yet the current implementation uses a fixed ratio; dynamic adjustment based on execution speed and coverage growth rate is a promising direction for future work.

6 Conclusion

We identify that the underperformance of learning-based fuzzers stems from an architectural mismatch rather than algorithmic limitations. Strategy effectiveness shifts at minute-level timescales, while existing in-loop architectures require tens of minutes per learning cycle, preventing timely adaptation regardless of which learning algorithm is used.

We propose TRIOFUZZ, a new fuzzing engine that decouples learning from execution through a three-tier thread architecture. A centralized Scheduler aggregates feedback from parallel Workers and Scouters, compressing the learning cycle to under one minute. Evaluation on FuzzBench and OSS-Fuzz shows that TRIOFUZZ achieves the highest coverage on 7 out of 9 FuzzBench targets and 8 out of 10 OSS-Fuzz projects. On Magma, TRIOFUZZ triggers 17% more vulnerabilities than AFL++ and achieves the fastest discovery on 22 out of 37 vulnerabilities. TRIOFUZZ further discovers 19 new CVEs in 6 real-world projects. Our ablation study confirms that the same learning algorithms perform better under the decoupled

architecture, demonstrating that architectural design is the key factor behind effective adaptive strategy selection in fuzzing.

Ethical Considerations

Stakeholder Identification. We identify the following stakeholders potentially impacted by our research process and its publication: (1) *Users of the fuzzed software.* Our fuzzing campaigns target open-source projects (Fio, Open Babel, Libucl, BehaviorTree.CPP, Ogre, Assimp) whose users may be exposed to the 19 vulnerabilities we discovered until patches are deployed. (2) *Open-source maintainers.* Maintainers receive vulnerability reports and bear the burden of developing and releasing patches under time pressure. (3) *The security research community.* Researchers benefit from the open-source release of TRIOFUZZ and the architectural insights in this paper, which can improve defensive testing infrastructure. (4) *Potential adversaries.* The publication of an effective fuzzing tool could lower the barrier for malicious actors to discover vulnerabilities in other software. (5) *The research team.* Extended fuzzing campaigns consumed significant computational resources, but posed no safety risks to team members.

Ethical Analysis. We structure our analysis around the Menlo Report principles [20].

Beneficence. The primary benefit of this work is improving defenders' ability to find vulnerabilities before adversaries do. TRIOFUZZ discovered 19 previously unknown CVEs across 6 projects, enabling maintainers to patch these issues proactively. The architectural insights generalize beyond our specific tool, potentially improving the effectiveness of the broader fuzzing ecosystem. We weigh these benefits against the dual-use risk that TRIOFUZZ could also be used offensively. We believe the net impact is positive: (a)TRIOFUZZ improves the efficiency of existing fuzzing techniques rather than enabling fundamentally new attack capabilities; (b) the vulnerabilities we found would likely be discovered by adversaries using existing tools given sufficient time; and (c) responsible disclosure ensures patches precede public knowledge.

Respect for Persons. Our evaluation primarily uses established public benchmarks (FuzzBench [15], OSS-Fuzz [14], Magma [17]) specifically designed for fuzzing research. No human subjects, user data, or live systems were involved. For real-world fuzzing of the 6 open-source projects, we tested only locally compiled binaries and did not interact with any deployed services or systems.

Justice. The benefits of our research (improved testing tools, patched vulnerabilities) are distributed broadly to the open-source community and end users, while the disclosure burden falls primarily on maintainers. To mitigate this asymmetry, we provided maintainers with detailed crash reports, proof-of-concept inputs, and root cause analysis to facilitate timely patching.

Respect for Law and Public Interest. All experiments were conducted on locally compiled open-source software. No unauthorized access to systems or services was performed.

Vulnerability Disclosure. We followed responsible disclosure practices for all 19 discovered vulnerabilities. Each vulnerability was reported to the respective project maintainers through their designated security channels or issue trackers before any public disclosure. Reports included detailed crash information, proof-of-concept inputs, and root cause analysis. All 19 vulnerabilities have been acknowledged by maintainers and assigned CVE identifiers.

Open Science

Our artifact is available at https://doi.org/10.5281/zenodo.20624509 and https://github.com/OldTruckDriver/triofuzz. The repositories contain the following artifacts:

Source Code. The complete implementation of TRIOFUZZ comprises approximately 35K lines of C++ code. This includes the three-tier thread architecture (Scheduler, Worker, and Scouter threads), the Pluggable Learning Module with default implementations of Thompson Sampling, Particle Swarm Optimization [29], and Markov Chain [21], and build scripts for compilation on Ubuntu 22.04. This artifact supports all results reported in Sections 4.2 to 4.4, 4.6 and 4.7.

Benchmark Infrastructure. Scripts and configurations to reproduce all experiments: (1) FuzzBench integration for the 9 targets evaluated in Section 4.2; (2) build scripts and harnesses for the 10 OSS-Fuzz projects evaluated in Section 4.3; (3) Magma benchmark configuration for the vulnerability detection experiments in Section 4.4;

Baseline Configurations. Docker containers for each baseline fuzzer (AFL++, LibFuzzer, MOpt, MuoFuzz, EcoFuzz, Honggfuzz) with the exact versions used in our evaluation, enabling fair comparison under identical environments.

Documentation. The repository includes installation instructions, usage examples, and a tutorial for extending TRIOFUZZ with custom learning algorithms through the Pluggable Learning Module interface defined in Algorithm 1.

Artifacts Not Shared. Proof-of-concept inputs for the 19 CVEs (Table 5) are withheld to prevent exploitation before patches are fully deployed. These will be released after a responsible disclosure period.

References

[1] libxml2: parser.h file reference. https://gnome.pages.gitlab.gnome.org/libxml2/html/parser_8h.html. Accessed: 2026-01-23. Documents that xmlCleanupParser is not thread-safe; also notes global-setting APIs like xmlSetExternalEntityLoader are not thread-safe.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[3] Omar Besbes, Yonatan Gur, and Assaf Zeevi. Stochastic multi-armed-bandit problem with non-stationary rewards. *Advances in neural information processing systems*, 27, 2014.

[4] Marcel Böhme, Valentin JM Manès, and Sang Kil Cha. 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*, pages 678–689, 2020.

[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.

[6] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[7] Yongheng Chen, Rui Zhong, Yupeng Yang, Hong Hu, Dinghao Wu, and Wenke Lee. {μFUZZ}: Redesign of parallel fuzzing using microservice architecture. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1325–1342, 2023.

[8] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. {EnFuzz}: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1967–1983, 2019.

[9] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX workshop on offensive technologies (WOOT 20)*, pages 1–N, 2020.

[10] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. Libafl: A framework to build modular and reusable fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1051–1065, 2022.

[11] Yu-Fu Fu, Jaehyuk Lee, and Taesoo Kim. autofz: Automated fuzzer composition at runtime. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1901–1918, 2023.

[12] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

[13] Google. honggfuzz, 2010. Accessed on August 4, 2025.

[14] Google. Oss-fuzz, 2016. Accessed on August 4, 2025.

[15] Google. Fuzzbench, 2025. Accessed on August 18, 2025.

[16] Emre Güler, Philipp Görz, Elia Geretto, Andrea Jemmett, Sebastian Österlund, Herbert Bos, Cristiano Giuffrida, and Thorsten Holz. Cupid: Automatic fuzzer selection for collaborative fuzzing. In *Proceedings of the 36th Annual Computer Security Applications Conference*, pages 360–372, 2020.

[17] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(3):1–29, 2020.

[18] Pooria Joulani, Andras Gyorgy, and Csaba Szepesvári. Online learning under delayed feedback. In *International conference on machine learning*, pages 1453–1461. PMLR, 2013.

[19] Siddharth Karamcheti, Gideon Mann, and David Rosenberg. Adaptive grey-box fuzz-testing with thompson sampling. In *Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security*, pages 37–47, 2018.

[20] Erin Kenneally and David Dittrich. The menlo report: Ethical principles guiding information and communication technology research. *Available at SSRN 2445102*, 2012.

[21] Konstantinos Kitsios, Marcel Böhme, and Alberto Bacchelli. On interaction effects in greybox fuzzing. *arXiv preprint arXiv:2510.19984*, 2025.

[22] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 2123–2138, 2018.

[23] Yuki Koike, Hiroyuki Katsura, Hiromu Yakura, and Yuma Kurogome. Slopt: Bandit optimization framework for mutation-based fuzzing. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 519–533, 2022.

[24] James Kukucka, Luís Pina, Paul Ammann, and Jonathan Bell. An empirical examination of fuzzer mutator performance. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1631–1642, 2024.

[25] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pages 475–485, 2018.

[26] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 533–544, 2019.

[27] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. Pafl: extend fuzzing optimizations of single mode to industrial parallel mode. In *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 809–814, 2018.

[28] LLVM Project. libFuzzer, 2024. Accessed on February 25, 2025.

[29] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, 2019.

[30] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019.

[31] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 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*, pages 1393–1403, 2021.

[32] Hyeonmin Mo, Jongmun Yang, and Yunho Kim. Rc-fuzzer: Recommendation-based collaborative fuzzer. *Journal of Systems and Software*, page 112564, 2025.

[33] Roberto Natella and Van-Thuan Pham. Profuzzbench: A benchmark for stateful protocol fuzzing. In *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*, pages 662–665, 2021.

[34] Sebastian Österlund, Elia Geretto, Andrea Jemmett, Emre Güler, Philipp Görz, Thorsten Holz, Cristiano Giuffrida, and Herbert Bos. Collabfuzz: A framework for collaborative fuzzing. In *Proceedings of the 14th European Workshop on Systems Security*, pages 1–7, 2021.

[35] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on security and privacy (SP)*, pages 615–632. IEEE, 2017.

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

[37] Wenxuan Shi, Hongwei Li, Jiahao Yu, Wenbo Guo, and Xinyu Xing. Bandfuzz: A practical framework for collaborative fuzzing with reinforcement learning. In *Proceedings of the 17th ACM/IEEE International Workshop on Search-Based and Fuzz Testing*, pages 55–56, 2024.

[38] Balazs Szorenyi, Róbert Busa-Fekete, István Hegedus, Róbert Ormándi, Márk Jelasity, and Balázs Kégl. Gossip-based distributed stochastic bandit algorithms. In *International conference on machine learning*, pages 19–27. PMLR, 2013.

[39] The FreeType Project. Face creation - freetype 2.14.1 api reference. https://freetype.org/freetype2/docs/reference/ft2-face_creation.html, 2025. Accessed: 2026-01-23. States that an FT_Face object can only be safely used from one thread at a time, and face creation/destruction with the same FT_Library must be single-threaded.

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

[41] Francesco Trovo, Stefano Paladino, Marcello Restelli, and Nicola Gatti. Sliding-window thompson sampling for non-stationary settings. *Journal of Artificial Intelligence Research*, 68:311–364, 2020.

[42] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V Krishnamurthy, and Nael Abu-Ghazaleh. {SyzVegas}: Beating kernel fuzzing odds with reinforcement learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2741–2758, 2021.

[43] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 511–522, 2013.

[44] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. One fuzzing strategy to rule them all. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1634–1645, 2022.

[45] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 2313–2328, 2017.

[46] Dongsong Yu, Yiyi Wang, Chao Zhang, Yang Lan, Zhiyuan Jiang, Shuitao Gan, Zheyu Ma, and Wende Tan. xfuzz: A flexible framework for fine-grained, runtime-adaptive fuzzing strategy composition. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):69–91, 2025.

[47] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. EcoFuzz: Adaptive Energy-Saving greybox fuzzing as a variant of the adversarial Multi-Armed bandit. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2307–2324. USENIX Association, August 2020.

[48] Michal Zalewski. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/, 2014.

[49] Anshunkang Zhou, Heqing Huang, and Charles Zhang. Kraken: Program-adaptive parallel fuzzing. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):274–296, 2025.

[50] Xu Zhou, Pengfei Wang, Chenyifan Liu, Tai Yue, Yingying Liu, Congxi Song, Kai Lu, Qidi Yin, and Xu Han. Ultrafuzz: Towards resource-saving in distributed fuzzing. *IEEE Transactions on Software Engineering*, 49(4):2394–2412, 2022.

[51] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.