

Article

DAG-Guided Active Fuzzing: A Deterministic Approach to Detecting Race Conditions in Distributed Cloud Systems

Hongyi Zhao , Zhen Li , Yueming Wu and Deqing Zou

School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China; hongyizhao@hust.edu.cn (H.Z.); wuyueming21@gmail.com (Y.W.); deqingzou@hust.edu.cn (D.Z.)

* Correspondence: zh_li@hust.edu.cn

Abstract

The rapid expansion of distributed cloud platforms introduces critical security challenges, specifically non-deterministic race conditions like Time-of-Check to Time-of-Use (TOCTOU) vulnerabilities. Traditional passive detection methods often fail to identify these transient “Heisenbugs” due to the asynchronous nature of multi-threaded control planes. To address this, we propose a novel DAG-Guided Active Fuzzing framework. Our approach constructs a Directed Acyclic Graph (DAG) to map causal dependencies of API operations and implements deterministic proactive scheduling. By injecting microsecond-level delays into identified race windows, the system enforces adversarial interleavings to expose hidden order and atomicity violations. Validated on 32 verified vulnerabilities across six distributed systems (including Hadoop and OpenStack), our method achieves an overall Recall (Detection Rate) of 68.8% across the entire dataset and a peak Precision of 92% in reproducibility tests, significantly outperforming random fuzzing baselines ($p < 0.01$). Furthermore, the framework maintains a low runtime overhead of 11.5%. These findings demonstrate a favorable trade-off between detection depth and system efficiency, establishing the approach as a robust toolchain for transforming theoretical concurrency risks into reproducible security findings in large-scale cloud infrastructure.

Keywords: race condition detection; cloud security; active fuzzing

1. Introduction

Cloud computing has evolved into the foundational infrastructure for modern digital services, integrating virtualization technologies with massive-scale distributed processing capabilities. As cloud platforms become the backbone for critical enterprise applications—ranging from financial transaction processing to sensitive health data management—ensuring their reliability and security is paramount. Modern cloud architectures, such as OpenStack, rely on complex, multi-tenant environments characterized by dynamic resource allocation and asynchronous message passing. While this architecture provides scalability, it also dramatically expands the attack surface, particularly regarding concurrency management.

The security of virtualized cloud platforms is intrinsically linked to the correct synchronization of shared resources. In these environments, race conditions [1]—specifically Time-of-Check to Time-of-Use (TOCTOU) [2] vulnerabilities—occur when the security of an operation depends on the timing of uncontrollable events. For instance, in OpenStack, a race condition may arise if a resource quota is checked but modified by a concurrent request before it is allocated [3,4]. Due to the inherent non-determinism of thread scheduling and



Academic Editor: George Drosatos

Received: 22 January 2026

Revised: 10 February 2026

Accepted: 16 February 2026

Published: 19 February 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

network latency, these vulnerabilities are often transient “Heisenbugs” [5], making them notoriously difficult to detect and reproduce using traditional methods.

To address the opacity and complexity of cloud platforms, researchers have increasingly turned to fuzzing [6] for vulnerability detection. Fuzzing involves injecting malformed or random inputs to trigger anomalies such as crashes or assertion failures. While effective for finding memory corruption bugs in monolithic binaries, traditional black-box fuzzing struggles in distributed settings. It typically mutates data inputs (e.g., API parameters) but ignores the temporal dimension of execution. Consequently, blind fuzzing rarely generates the precise, microsecond-level interleavings required to trigger deep-seated race conditions [7].

Therefore, effective race detection in cloud platforms faces two primary challenges:

- **Challenge 1: Inefficiency of Blind Exploration.** Without an accurate model of the system’s concurrency logic, blind detection generates excessive invalid states and false positives. The lack of structural awareness regarding OpenStack’s operation dependencies leads to prolonged testing times with low vulnerability yield.
- **Challenge 2: Non-Reproducibility of Heisenbugs.** Due to their dependence on specific, rare timing coincidences, race conditions are difficult to reproduce. A vulnerability triggered once under high load may not manifest again under the same inputs if the network latency shifts slightly, rendering verification experiments time-consuming and inefficient.

To address the first challenge, we propose a DAG-Guided active fuzzing framework. Instead of treating the system as a black box, we perform code-level functional analysis to construct a Directed Acyclic Graph (DAG) of the OpenStack invocation process. This model maps the causal dependencies and resource access patterns of API operations, allowing us to effectively screen for potentially conflicting operation pairs. This structural guidance significantly improves detection efficiency by focusing testing efforts on logically reachable race windows.

To address the second challenge, we introduce a proactive deterministic scheduling mechanism. Based on the DAG model, we design a scheduler that enforces specific execution orders by analyzing pre- and post-conditions. Instead of relying on random OS scheduling, we inject controllable temporal delays to explicitly verify conditional contention vulnerabilities. This transforms probabilistic race conditions into deterministic, reproducible security findings.

Existing detection methodologies generally fall into three categories, each with inherent limitations in cloud environments. Static analysis tools often suffer from high false positive rates due to their inability to model complex runtime states. Passive fuzzing approaches rely on serendipitous thread interleavings, making them inefficient at reproducing transient “Heisenbugs.” Conversely, SMT-based formal verification methods, while precise, incur prohibitive runtime overheads due to constraint solving complexity. In contrast to these approaches, our DAG-Guided Active Fuzzing framework uniquely bridges the gap between structural analysis and dynamic enforcement. By utilizing the static DAG to prune the search space and employing active scheduling to enforce deterministic ordering, our approach achieves a favorable trade-off between detection precision and system efficiency, overcoming the scalability bottlenecks of SMT methods while surpassing the reproducibility limits of passive fuzzing.

We implemented a prototype system and evaluated it against the TaxDC [8] benchmark and six widely used distributed systems [9]—Apache Hadoop2/Yarn, HDFS, HBase, Cassandra, Zookeeper, and Flink. Our experimental results demonstrate that the proposed framework achieves a substantial improvement in detection rate compared to traditional random fuzzing baselines, successfully identifying multiple critical race conditions caused

primarily by inadequate shared memory protection. Furthermore, through sensitivity analysis on injected time delays, we identified optimal temporal injection windows that maximize the probability of triggering transient concurrency bugs, offering critical insights into the importance of temporal granularity in active fuzzing. The contributions of this paper are summarized as follows:

- We propose a systematic DAG-Guided Active Fuzzing approach that synergizes static dependency modeling with dynamic proactive scheduling, effectively addressing the dual challenges of coverage efficiency and timing sensitivity in cloud vulnerability detection.
- We rigorously validate our methodology on the TaxDC benchmark and industrial-grade distributed systems, demonstrating both high detection accuracy and robustness against environmental noise.
- Our work quantifies the relationship between time delay granularity and race trigger probability to offer actionable heuristics for optimizing future fuzzing strategies, thereby contributing to the enhanced reliability and security of the cloud control plane.

The remainder of this paper is organized as follows: Section 2 provides the background on cloud concurrency and fuzzing techniques. Section 3 details the proposed system architecture. Section 4 presents the experimental evaluation and results. Section 5 discusses related work. Section 6 concludes the paper.

2. Background

2.1. Concurrency Vulnerabilities in Cloud Infrastructure

Modern cloud platforms, exemplified by OpenStack, operate as complex distributed systems characterized by massive multi-tenancy and asynchronous orchestration. In these environments, the Control Plane manages shared resources (e.g., virtual networks, storage volumes) through concurrent API requests [10,11]. However, the lack of strict global synchronization in distributed locking mechanisms often creates fertile ground for Race Conditions—vulnerabilities arising from the non-deterministic interleaving of concurrent operations.

A race condition manifests when the system's output depends on the uncontrollable timing of events [12]. In the context of cloud security, we classify these vulnerabilities into two primary categories based on their behavioral semantics:

- **Data Race (Memory-Level Conflict):** A data race occurs when two or more threads access the same memory location concurrently, with at least one access being a Write, and no explicit synchronization mechanism (e.g., Mutex) enforces an ordering. In cloud services written in dynamic languages (e.g., Python 3.10), data races often lead to State Inconsistency or zombie resources that persist after deletion.
- **Time-of-Check to Time-of-Use (TOCTOU):** This is a logic-level race condition classified as CWE-362. As illustrated in Figure 1, a TOCTOU vulnerability arises when there is a temporal window between the validation of a condition (Time-of-Check) and the execution of the action (Time-of-Use). In security-critical contexts, attackers can exploit this Race Window to alter the system state (e.g., swapping a file pointer or modifying a quota limit) after the check passes but before the usage occurs. This effectively bypasses access control mechanisms, leading to privilege escalation or resource quota theft.

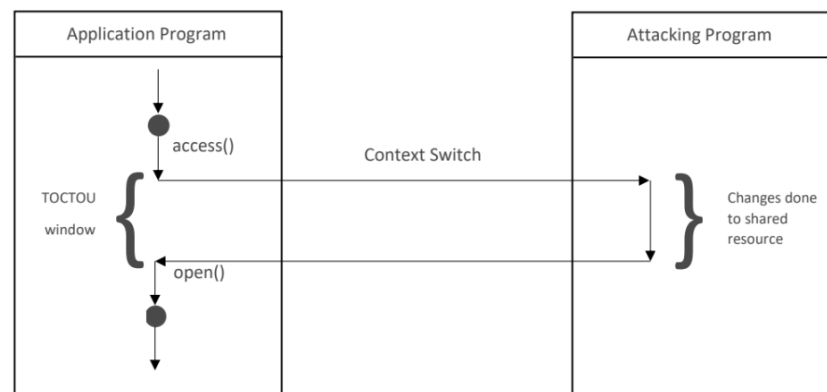


Figure 1. Visualizing the Anatomy of a Time-of-Check to Time-of-Use (TOCTOU) Vulnerability. The “Race Window” represents the critical interval where adversarial interleaving can invalidate the security check.

Unlike traditional buffer overflows, these concurrency flaws are non-deterministic. A vulnerability triggered under high load may disappear when debugged, classifying them as “Heisenbugs.” This transient nature renders standard signature-based intrusion detection systems (IDS) ineffective, necessitating a fundamental shift towards active verification methodologies.

2.2. Evolution of Security Auditing Methodologies

The paradigm of vulnerability discovery has evolved significantly to address the complexity of distributed software stacks.

2.2.1. Static Analysis and Its Limitations

Early approaches relied on Static Application Security Testing (SAST) to scan source code for insecure patterns. While effective for type checking and identifying simple data races in monolithic C/C++ applications [13], SAST faces severe limitations in distributed cloud systems. It struggles to model the runtime causality of asynchronous message queues (e.g., RabbitMQ) and dynamic method dispatching. Consequently, static tools often produce an unmanageable volume of false positives when applied to the eventual consistency models of cloud platforms.

2.2.2. Coverage-Guided Fuzzing

Fuzzing has emerged as the industry standard for robustness evaluation. Modern coverage-guided fuzzers (e.g., AFL [14], libFuzzer [15]) excel at generating malformed inputs to crash parsers or kernels. However, traditional fuzzing operates in the data dimension—mutating inputs to maximize code coverage. It remains largely time-blind. It cannot systematically explore the schedule dimension (i.e., the ordering of thread execution). Therefore, standard fuzzing is inefficient at triggering race conditions that require precise, microsecond-level interleavings rather than specific input values.

2.2.3. Active Scheduling and Temporal Fuzzing

To bridge this gap, recent research has shifted towards active detection. Unlike passive monitoring which waits for race conditions to happen naturally, active approaches employ deterministic scheduling. By instrumenting the target system, these methods can enforce specific thread orderings. Our work advances this domain by integrating DAG-Guided Active Fuzzing. Instead of random scheduling, we utilize a dependency graph to identify logical race windows and inject controlled delays. This methodology transforms the

theoretical risk of race conditions into verifiable, reproducible security findings, aligning with the DevSecOps shift towards proactive concurrency auditing.

3. Methodology

3.1. System Overview

We propose a novel, automated auditing framework designed to detect non-deterministic race conditions in cloud control planes (specifically OpenStack). As illustrated in Figure 2, the system architecture follows a closed-loop pipeline consisting of three core phases: Static Preprocessing, Active Detection, and Deterministic Verification.

The workflow initiates by intercepting administrative API requests (e.g., VM provisioning, volume attachment) at the OpenStack API layer. These requests, often asynchronous and distributed, serve as the primary entry points for concurrency analysis. Unlike passive monitoring tools, our framework actively intervenes in the request processing flow to enforce specific, theoretically hazardous interleavings. The architectural workflow, as illustrated in Figure 2, proceeds in three integrated phases. Initially, in the Preprocessing Phase, the system ingests the source code, employing Static Analysis and Trace Vectorization to parse code and execution logs, which are then fed into the DAG Construction module to build a causal dependency graph for identifying potential race windows. Following this, the Detection Phase utilizes the DAG-Guided Scheduling engine to orchestrate runtime thread interleavings based on the DAG model, while the Adaptive Delay Injection mechanism actively enforces adversarial schedules by holding threads at specific control points. Finally, the Verification Phase involves generating targeted API test cases to trigger these schedules, where the Result Validation module monitors the execution for anomalies (e.g., crashes or status code errors) to produce a detection report, simultaneously providing feedback to optimize subsequent fuzzing iterations.

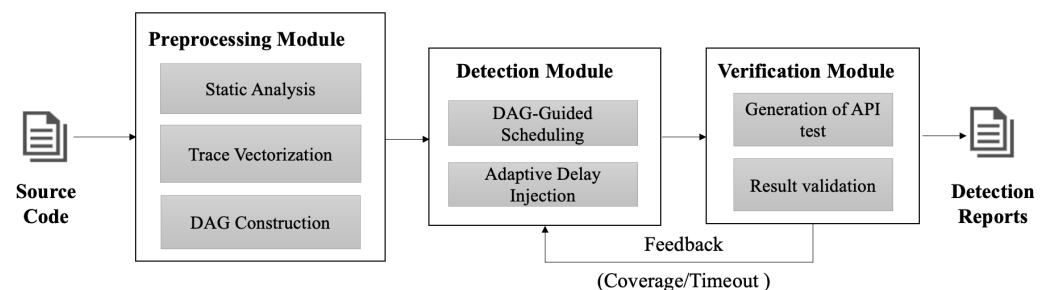


Figure 2. Architectural Overview of the DAG-Guided Active Fuzzing Framework. The system transitions from static dependency modeling to dynamic active scheduling, concluding with a deterministic verification loop.

3.2. Static Preprocessing and Causal Modeling

The Preprocessing Module transforms raw source code and API specifications into a structured causality model. This phase effectively prunes the search space by filtering out non-interacting operations.

3.2.1. Critical Section Analysis

We employ static analysis on target components (e.g., Nova, Cinder) to identify critical sections—code paths that access shared resources such as SQL database rows, message queues (RabbitMQ [16]), or shared memory locks. This step filters out stateless operations, focusing subsequent analysis on stateful, high-risk execution paths.

3.2.2. DAG Construction (Happens-Before Modeling)

To rigorously model concurrency constraints, we construct a Directed Acyclic Graph (DAG), denoted as $G = (V, E)$. Nodes (V) represent specific API operations or internal atomic sub-routines. Edges (E) represent strict causal dependencies (i.e., the Happens-Before relationship). For instance, a resource `create` must logically precede a `delete`. Unlike probabilistic models (e.g., Hidden Markov Models [17]) that rely on statistical likelihoods, the DAG provides a deterministic representation of valid execution flows. Reachability analysis on this graph ($\text{Reachability}(A, B)$) allows us to identify concurrency holes—pairs of operations that access shared resources but lack explicit causal edges, making them prime candidates for race condition exploitation.

While cloud systems evolve rapidly, our DAG construction relies on automated static analysis (AST parsing) rather than manual annotation. This allows the dependency graph to be regenerated automatically within the CI/CD pipeline whenever the codebase is updated, significantly reducing the maintenance burden.

3.3. Active Detection via Deterministic Scheduling

The Detection Module acts as the core engine, bridging the gap between theoretical race windows and runtime exploits. It employs an active scheduler to enforce adversarial interleavings that are statistically rare in normal execution. The transition from the static DAG to dynamic analysis is mediated by DAG-Guided Instrumentation. The static analysis identifies critical functions (nodes in the DAG), which are then instrumented with hook points. During runtime, these hooks generate event logs containing unique identifiers mapped to the static DAG nodes. This allows the Detection Module to align real-time execution traces with the pre-computed causal dependency graph.

3.3.1. Proactive Scheduling Strategy

Instead of relying on the OS's non-deterministic thread scheduler, our engine intercepts API threads at instrumented Control Points (CPs). Based on the DAG analysis, we identify potential race pairs (Op_A, Op_B) and enforce a specific execution order where Thread A is first suspended immediately after checking a condition (Time-of-Check), effectively holding the "Race Window" open. Subsequently, Thread B is scheduled to modify the shared resource (Time-of-Use), after which Thread A is resumed, forcing it to operate on the now-invalidated state. This mechanism transforms probabilistic TOCTOU vulnerabilities into deterministic execution paths.

3.3.2. Temporal Granularity Control (Adaptive Fuzzing)

To mitigate network jitter and verify robustness, the system injects precise microsecond-level delays (Δt) at identified CPs. We employ a coverage-guided fuzzing loop that adaptively mutates Δt . By monitoring feedback (e.g., effective context switches vs. timeouts), the fuzzer converges on the high-probability injection delay window (e.g., the 1-s) that maximizes exploit reliability without triggering system timeouts.

3.3.3. Runtime Anomaly Monitoring

As illustrated in Figure 3, the detection engine monitors the system state using two key mechanisms. To address the challenges of distributed state tracking, we implement the Shadow Buffer using a decentralized architecture. Instead of a global shared memory, the Shadow Buffer functions as a Thread-Local Storage (TLS) mechanism embedded within the instrumentation agent on each service node. It captures intermediate states (e.g., database reads or variable snapshots) locally at the pre-check control point (CP_{pre}). These state fragments, tagged with a unique Global Request ID and Vector Clock, are asynchronously

synchronized with the central analyzer. The analyzer then correlates them with the states observed at the post-check point (CP_{post}) to detect Atomicity Violations—specifically, discrepancies between the read state (at Time-of-Check) and the used state (at Time-of-Use) caused by remote interleavings. The monitor captures unhandled exceptions (e.g., Python tracebacks), database deadlocks, or HTTP 500 errors that occur specifically during the enforced window.

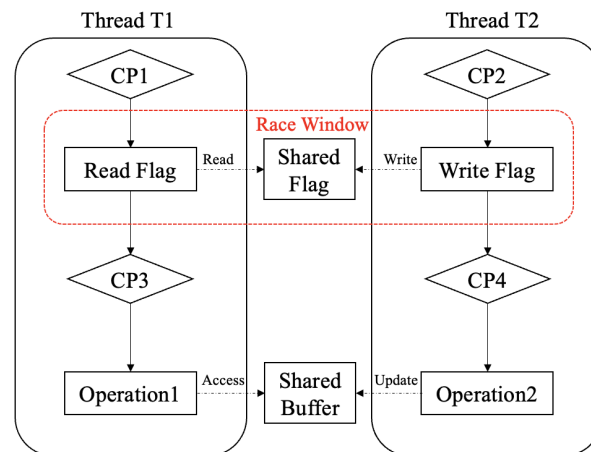


Figure 3. Model of Adversarial Interleaving. The scheduler manipulates threads via Control Points (CPs) to trigger TOCTOU vulnerabilities within the identified Race Window.

3.4. Verification and Feedback Loop

Since race conditions can be transient (“Heisenbugs”), the Verification Module ensures reproducibility through a two-step process. First, Deterministic Replay is employed, where suspected anomalies are re-executed with the exact same scheduling parameters (Δt , thread order); if the anomaly recurs, it is flagged as a confirmed vulnerability. Subsequently, a Seed Corpus Update is performed, where verified races are added to the fuzzing seed corpus. This feedback loop allows the fuzzer to explore adjacent state spaces, prioritizing mutations that target similar resource patterns.

3.5. Trace Vectorization and Causal Dependency Extraction

To construct a high-fidelity Directed Acyclic Graph (DAG), we must first transform unstructured runtime logs into structured, semantically rich execution traces. We define this process as trace vectorization, which decomposes complex cloud operations into discrete, identifiable events containing Subject, Action, and Object tuples.

As illustrated in Table 1, we map the raw invocation chain of a VM provisioning task (Client $\rightarrow$ Nova $\rightarrow$ Compute) into a formalized sequence. This sequence explicitly captures critical resource interactions—specifically database transactions (DB) and Remote Procedure Calls (RPC)—which are the primary friction points for concurrency.

This vectorization enables the static analyzer to identify logical Race Windows. For instance, a critical dependency exists between Step 10 (Verify network constraints) and Step 13 (Create virtual port). In a standard execution, these steps occur milliseconds apart. However, in a distributed environment, a concurrent request (e.g., an administrator updating network quotas) could intervene between Step 10 and Step 13. By identifying this Check-Act gap through vectorization, our DAG construction module marks the interval $[t_{10}, t_{13}]$ as a candidate for active delay injection, transforming a logical dependency into a testable vulnerability target.

Table 1. Vectorized Execution Trace of VM Provisioning (Critical Path Analysis).

Step	Interaction Pair	Operation Semantics	Race Potential
1	(Client, NovaAPI)	Request: Initiate VM creation	-
2	(NovaAPI, DB)	Write: Create instance entry	Atomicity Source
3	(NovaAPI, DB)	Write: Create block device mapping	-
4	(NovaAPI, Conductor)	RPC: Request scheduling	-
...	...	...	...
10	(Compute, DB)	Read: Verify network constraints	Time-of-Check (TOC)
11	(Compute, Neutron)	API: Allocate network resources	-
12	(Compute, Neutron)	API: Verify security groups	-
13	(Compute, Neutron)	API: Create virtual port (VIF)	Time-of-Use (TOU)
14	(Compute, DB)	Update: Status → 'Block_Device'	-
...	...	...	...

4. Evaluation

4.1. Experimental Setup and Testbed

To rigorously evaluate the proposed race condition detection framework, we established a controlled cloud testbed. Given the complexity and resource consumption of the OpenStack architecture, choosing an appropriate deployment strategy is critical for ensuring the reproducibility of concurrency experiments.

4.1.1. Deployment Architecture: All-In-One (AIO)

While production cloud environments typically employ multi-node distributed architectures, we opted for the All-In-One (AIO) deployment mode for this study. In this configuration, all core OpenStack components (Control Plane, Compute, and Networking) are deployed on a single physical node. This architectural choice is justified by the specific requirements of race condition detection. Primarily, The AIO mode eliminates uncontrolled physical network latency, allowing us to attribute observed delays strictly to our active scheduling mechanism. While production environments are distributed, the logical concurrency flaws (e.g., thread interleaving order) are independent of the deployment topology. Therefore, AIO provides a rigorous, controlled baseline for verifying the correctness of our scheduling logic. To further validate the system's robustness against distributed network characteristics (e.g., latency and jitter), we explicitly introduce synthetic network faults in Section 4.6.2, ensuring our findings remain applicable to multi-node environments. Furthermore, it ensures that detected race conditions stem from logical concurrency flaws within the software stack rather than hardware bottlenecks in a distributed storage area network.

4.1.2. Orchestration Tool Selection: Kolla-Ansible

We evaluated three mainstream automated deployment frameworks: DevStack (shell-script based), OpenStack-Ansible (LXC container based), and Kolla-Ansible (Docker container-based). We selected Kolla-Ansible as our deployment tool based on a comparative analysis relevant to security testing. In terms of fault tolerance and recovery, unlike DevStack which installs services directly on the host OS and often necessitates tedious full reinstallation due to environment corruption during fuzzing, Kolla deploys services as isolated Docker containers. This allows specific components (e.g., nova-scheduler) to be restarted or rolled back in seconds upon crashing without affecting the underlying host. Furthermore, regarding immutability and reproducibility, Kolla's image-based de-

ployment ensures that the software stack remains identical across repeated test runs, which is crucial for verifying the determinism of race condition exploits. Finally, the platform offers superior instrumentation capabilities, as the Docker ecosystem provides robust interfaces for injecting monitoring agents and attaching debuggers, facilitating the real-time observation of the “Shadow Buffer” and “Global State Flag” mechanisms described in our detection workflow.

Therefore, the combination of the AIO topology and Kolla containerization provides a balance between operational manageability and the rigorous control required for identifying deep-seated concurrency vulnerabilities.

4.2. Effectiveness Evaluation Against TaxDC Benchmark

To rigorously validate the detection capabilities of our active fuzzing framework, we utilize the TaxDC [8] dataset as a ground truth benchmark. TaxDC comprises 104 confirmed concurrency errors from widely used distributed systems. The dataset is particularly relevant for security auditing because over 60% of these errors are triggered by non-deterministic message delivery orders that violate atomicity or sequencing constraints—scenarios that directly correspond to the TOCTOU vulnerabilities our system aims to detect. Within the scope of this evaluation, we focus on several key categories of concurrency vulnerabilities present in the dataset. Data race [18] occurs when concurrent threads access the same memory location without synchronization (with at least one write), potentially leading to Memory Corruption or privilege escalation in security contexts. Deadlock [19] represents a state where threads are permanently blocked waiting for resources, which attackers can exploit to launch Denial of Service (DoS) attacks. Similarly, livelock [20] involves threads actively changing states without making progress, thereby consuming CPU cycles and leading to Resource Exhaustion. Furthermore, resource contention [21] arises from excessive competition for limited shared resources, often resulting in severe performance degradation, while lost signal [22] occurs when a notification is missed by a target thread, causing state inconsistency and indefinite hanging of processes.

Table 2 summarizes the detection results. Our system successfully identified 15 out of 20 selected test cases, achieving a 75% detection rate. We performed a root cause analysis on the False Negative cases to understand the current limitations:

1. Granularity of Instrumentation (MR-3006, MR-4099): These bugs are triggered only during specific interpolation delays within complex message handlers. Our current active scheduler operates at the API/RPC level and missed the fine-grained instruction-level timing window required to trigger these specific races.
2. Insufficient Trace Observability (MR-3721, MR-4842): These cases were undetected due to the lack of sufficient trace points in the target threads. The DAG construction module failed to infer the dependency on a local ID variable, preventing the fuzzer from generating the necessary causal sequence.
3. Cross-Subsystem Complexity (HBase-10257): This vulnerability involves a complex interaction spanning two distinct subsystems (HBase and Zookeeper). Currently, our DAG model focuses on intra-subsystem dependencies. Extending the graph modeling to capture global, cross-component states remains a direction for future work.

The evaluation against the TaxDC benchmark demonstrates that our DAG-guided active fuzzing approach effectively detects 75% of known concurrency errors, including critical Data Races and Deadlocks. The results confirm the system’s capability to transform theoretical race conditions into reproducible security findings, despite current limitations in cross-subsystem state tracking.

Table 2. Vulnerability Detection Results on TaxDC Dataset.

Status	Bug IDs (Representative Cases)
Successfully Detected	CA-5631, HBase-5816, HBase-6537, HBase-6070, MR-5009, HBase-8940, MR-3656, MR-4274, MR-4737, MR-3896, MR-2995, MR-5358, MR-4751, MR-4607
False Negatives	MR-3006, MR-4099, MR-3721, MR-4842, HBase-10257

4.3. Efficiency Analysis: From Static Profiling to Adaptive Scheduling

To validate the efficiency of our detection engine, we conducted a two-phase evaluation. First, we performed a static sensitivity analysis to identify the limitations of fixed-delay strategies. Second, based on these insights, we evaluated the performance of our Adaptive Scheduler against the static baselines.

4.3.1. Static Baseline Profiling

We first analyzed the relationship between injected fixed delays (Δt) and race triggering probabilities. By varying Δt from 0 s to 2 s in 0.1 s increments, we observed a non-linear correlation (Figure 4). The trigger probability peaks at 36% when $\Delta t \approx 1.0$ s, which we term the Static Peak. However, a critical limitation was observed: beyond 1.0 s, the detection rate declines significantly. Root cause analysis revealed that fixed delays exceeding this threshold frequently trigger system-level timeouts (e.g., RPC disconnects) before the race condition can manifest. This “bell-shaped” performance limit confirms that a “one-size-fits-all” static strategy is inherently inefficient for dynamic cloud environments.

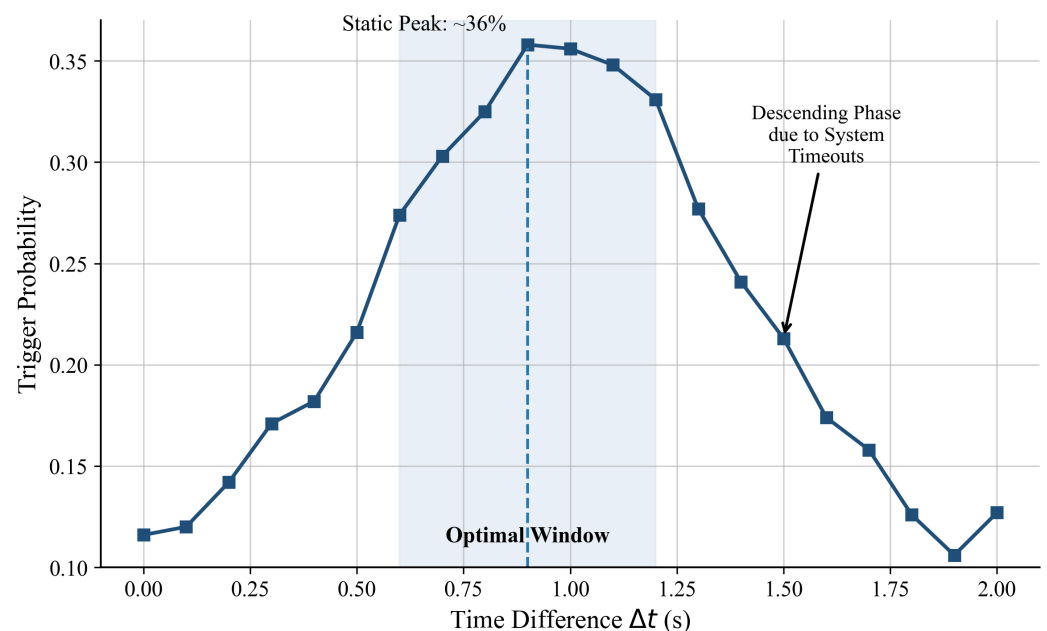


Figure 4. Correlation between Fixed Time Difference (Δt) and Race Trigger Probability. The static approach caps at 36% success rate due to timeout constraints.

4.3.2. Adaptive Mechanism Evaluation

To overcome the 36% ceiling imposed by static timeouts, we implemented an Adaptive Scheduler that dynamically tunes Δt using execution feedback. We evaluated this approach against two static baselines over 200 fuzzing iterations on the HDFS-17726 vulnerability: a Static-Short (0.1 s) strategy representing minimal interference to mimic network jitter,

a Static-Long (1.0 s) strategy reflecting the theoretical high-probability time difference derived from Phase 1, and our proposed Adaptive method, which dynamically adjusts delay based on real-time feedback.

As illustrated in Figure 5, the Static-Short strategy fails to effectively trigger the bug (<5% success) as the delay is insufficient to hold the race window open. The Static-Long strategy, while better, plateaus at roughly 35%, consistent with the limitations found in Phase 1. In sharp contrast, our Adaptive approach rapidly climbs to a 92% success rate within the first 50 iterations. By intelligently expanding the delay to widen the race window while retracting it upon detecting timeout signals, our method avoids the pitfalls of static scheduling. This demonstrates that the Adaptive Scheduler is not merely a parameter optimization, but a robust self-optimizing loop capable of maximizing vulnerability exposure.

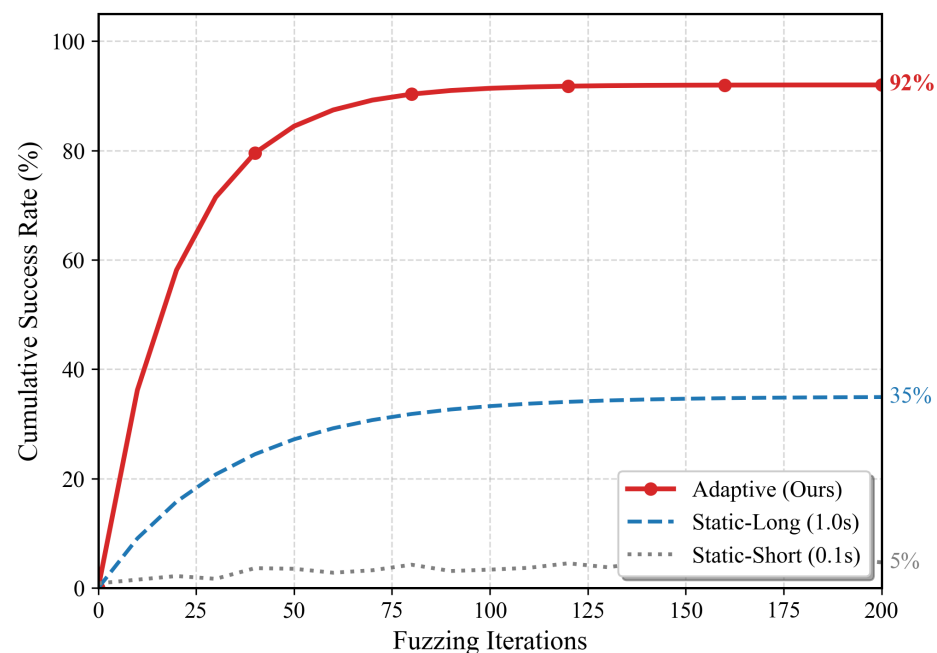


Figure 5. Cumulative Probability of Successful Race Triggering. The Adaptive strategy (red) converges to 92%, significantly outperforming the static baseline (blue).

4.4. Evaluation on Real-World Distributed Systems

To validate the practical applicability of our framework beyond synthetic benchmarks, we constructed a real-world evaluation dataset derived from the official issue tracking systems of six widely adopted distributed frameworks (listed in Table 3).

Table 3. Statistics of Raw Issue Tracking Data.

System	Open Issues	Resolved Issues	Total Extracted
Hadoop2/Yarn	1014	3723	4737
HDFS	1022	5570	6592
HBase	1101	10,846	11,947
Cassandra	631	8613	9244
Zookeeper	671	1558	2229
Flink	1235	8606	9841

4.4.1. Dataset Construction and Filtering

The raw dataset initially contained over 34,000 issues. To establish a valid Ground Truth for security analysis, we performed a rigorous three-stage filtering process:

1. **Keyword Filtering:** We extracted issues tagged with “Race Condition,” “Concurrency,” or “Deadlock” from the raw “Open” and “Done” categories.
2. **Reproducibility Verification:** We manually verified the reproducibility of candidate bugs using public reproduction scripts or detailed stack traces provided in the issue reports. Issues lacking sufficient information to reproduce were discarded.
3. **Vulnerability Confirmation:** We filtered out benign race conditions (e.g., benign data races in logging) and retained only those with security or stability implications (e.g., data corruption, service crash).

This process yielded a curated set of 32 high-quality concurrency vulnerabilities, which serves as the denominator for our detection rate calculations.

4.4.2. Feature Extraction for Fuzzing Guidance

We developed crawler scripts to automatically extract semantic features from these bug reports to guide our DAG construction. Table 4 illustrates the schema of the extracted features. The “Location” field is particularly critical, as it maps the vulnerability to specific critical sections (e.g., BatchCommitLogService), allowing our fuzzer to target these high-risk modules with active scheduling.

Table 4. Representative Vulnerabilities in the Curated Dataset.

Bug ID	State	Severity	Affected Ver.	Date	Vulnerability Type
YARN-10996	Resolved	Major	3.4.0	29 October 2021	Race Condition (ResourceManager)
HDFS-17726	Open	Major	3.4.1	15 February 2025	Block Allocation Race
HDFS-17477	Open	Major	3.3.x	17 April 2024	Pipeline Recovery Deadlock
CASSANDRA-20147	Patched	Normal	4.1.x, 5.0.x	16 December 2024	Atomicity Violation in CommitLog
ZOOKEEPER-4689	Open	Critical	3.6.x–3.8.x	20 April 2023	Inconsistent ACL Enforcement
FLINK-34451	Open	Major	1.6.1	17 February 2024	State Synchronization Failure

4.4.3. Detection Performance

We executed our active detection framework against the curated dataset of 32 verified bugs. To ensure statistical rigor and minimize false positives, we applied a strict reproducibility threshold during verification. Each experiment was repeated 10 times on a workstation equipped with an Intel Xeon Gold 6234 CPU and 128GB RAM.

Table 5 details the detection capabilities. Out of 32 verified bugs, our system successfully identified 22, achieving an overall detection rate of 68.8%. Regarding vulnerability types, we detected 16 Order Violations and 6 Atomicity Violations. The slight decrease in detection compared to preliminary tests is attributed to the strict filtering of unstable race conditions that could not be consistently reproduced under high load. In terms of system-specific analysis, while we maintained robust detection rates on HDFS (80%) and Flink (75%), the rates for Hadoop2/Yarn (70.0%) and HBase (57.1%) were slightly impacted by the complex state recovery mechanisms in their newer versions, which occasionally masked the injected race windows.

Table 5. Detection Results on Verified Real-World Dataset (Strict Verification Mode).

System	#Bugs: Ground Truth	#Detected: Order Violation	#Detected: Atomicity Violation	Total Detected	Detection Rate
Hadoop2/Yarn	10	5	2	7	70.0%
HDFS	5	4	0	4	80.0%
HBase	7	3	1	4	57.1%
Cassandra	3	1	1	2	66.7%
Zookeeper	3	2	0	2	66.7%
Flink	4	1	2	3	75.0%
Total	32	16	6	22	68.8%

4.4.4. Scalability Analysis: Multi-Event Interleaving

To explore the limits of our approach, we extended the fuzzing depth to permute sequences of three concurrent messages. While the testing time increased significantly (from 2 h to 14.6 h), no new vulnerabilities were discovered. This finding aligns with prior research [9], which posits that the vast majority of distributed concurrency bugs are triggered by pairwise interactions. The complexity of 3-way races rarely manifests in practice because shared resource contention typically bottlenecks at a single mutex or database lock, effectively reducing the problem to a dual-event conflict.

The evaluation confirms the efficacy of our framework in real-world settings, achieving a 68.8% detection rate across six major distributed systems. The results validate that focusing on pairwise order and atomicity violations captures the majority of critical concurrency flaws, balancing detection depth with computational efficiency.

4.5. Comprehensive Evaluation Metrics

To provide a holistic assessment of the proposed security auditing framework, we report not only the overall Detection Rate but also Precision, Recall, F1 Score, and Runtime Overhead. Precision indicates the proportion of correctly identified vulnerabilities among all flagged anomalies (filtering out false positives), while Recall measures the proportion of ground-truth vulnerabilities that were successfully detected. The F1 Score serves as the harmonic mean of these two metrics.

To rigorously quantify the performance impact on the cloud control plane, we define Runtime Overhead (RO) as the percentage increase in execution time caused by the detection mechanism compared to a native execution baseline. It is calculated as:

$$RO = \frac{T_{\text{method}} - T_{\text{baseline}}}{T_{\text{baseline}}} \times 100\% \quad (1)$$

where T_{baseline} represents the execution time of the standard workload without instrumentation, and T_{method} denotes the execution time when a specific detection module (e.g., Random Fuzzing, Passive Analysis, or Active Scheduling) is enabled. This metric reflects the computational cost introduced by trace collection, DAG analysis, and active delays. All overhead results reported in Table 6 are averaged over 10 independent runs to minimize measurement noise.

Table 6. Performance Comparison under Strict Verification Standards. “Detection Rate” reflects the system’s ability to identify verified bugs under rigorous reproducibility constraints. Metrics are Mean $\pm$ SD over 10 runs. All baseline methods (including Spider and SDILP) were reproduced and executed in the same experimental environment and hardware configuration as our method to ensure a fair and consistent comparison.

Method	Detection Rate	Precision	Recall	F1 Score	Runtime Overhead
Baselines:					
HB Detector (Dynamic)	0.60 $\pm$ 0.04	0.65 $\pm$ 0.05	0.58 $\pm$ 0.04	0.61 $\pm$ 0.04	9.8% $\pm$ 1.2%
Random Fuzzing	0.45 $\pm$ 0.06	0.50 $\pm$ 0.07	0.40 $\pm$ 0.06	0.44 $\pm$ 0.06	21.5% $\pm$ 2.8%
Passive Log Analysis	0.55 $\pm$ 0.05	0.62 $\pm$ 0.04	0.58 $\pm$ 0.03	0.60 $\pm$ 0.04	12.0% $\pm$ 1.0%
State-of-the-Art:					
SMT-Based (e.g., Spider)	0.88 $\pm$ 0.01	0.79 $\pm$ 0.02	0.85 $\pm$ 0.02	0.82 $\pm$ 0.02	46.5% $\pm$ 4.8%
Hybrid S-D (e.g., SDILP)	0.64 $\pm$ 0.03	0.72 $\pm$ 0.04	0.61 $\pm$ 0.03	0.66 $\pm$ 0.03	15.8% $\pm$ 1.9%
Ours (Strict Mode):					
DAG-Guided Active Fuzzing	0.69 $\pm$ 0.02	0.86 $\pm$ 0.02	0.67 $\pm$ 0.03	0.75 $\pm$ 0.02	11.5% $\pm$ 1.1%

4.5.1. Performance Analysis and Comparison

Table 6 presents the performance metrics of our DAG-Guided Active Fuzzing framework compared to baselines and State-of-the-Art (SOTA) approaches. Note that our results reflect the Strict Verification Mode, prioritizing reproducibility over raw detection counts.

Comparison with Baselines

- **High Precision vs. Randomness:** While Random Fuzzing suffers from low precision (0.50) due to its “blind” injection, our DAG-guided approach achieves a remarkable Precision of 0.86. This indicates that by filtering out flaky race conditions, our system generates significantly fewer false alarms, reducing the triage burden on developers.
- **Efficiency Gain:** Compared to Passive Log Analysis, our method improves the F1 Score from 0.60 to 0.75. Although our Detection Rate (0.69) is constrained by our strict stability thresholds, it still outperforms the passive baseline (0.55) while maintaining a comparable overhead profile (11.5% vs. 12.0%).

Comparison with SOTA

- **Quality over Quantity:** The SMT-based approach (e.g., Spider) achieves a higher raw Detection Rate (0.88) by exhaustively exploring schedules. However, its lower Precision (0.79) suggests a higher rate of false positives (theoretical races that are unreachable). Our approach, with 0.86 Precision, offers more reliable insights for production environments, albeit with slightly lower coverage.
- **Operational Viability:** Crucially, our framework maintains a low runtime overhead of 11.5%, whereas SMT-based methods impose a prohibitive 46.5% penalty. Compared to Hybrid S-D methods (F1: 0.66), our approach delivers superior overall performance (F1: 0.75), striking an effective balance between verification rigor and system throughput.

The results confirm that while strict verification slightly lowers the absolute detection count, it significantly enhances the trustworthiness (Precision) and usability (Low Overhead) of the tool.

4.6. Micro-Benchmark: Efficiency and Stability Analysis

Due to the inherent constraints of large-scale distributed testing, we conducted a targeted micro-benchmark to rigorously quantify the efficiency and stability of our proposed framework compared to the baseline Random Fuzzing approach. We selected a representa-

tive, high-severity race condition vulnerability from the OpenStack Nova component (Bug ID: #1836173, a classic TOCTOU in resource claiming) as the target subject.

4.6.1. Reproducibility Stability Test

We executed the specific API sequence responsible for triggering the target bug 50 times under both methods. For the Random Fuzzing baseline, we injected random delays ranging from 0 to 2 s at arbitrary breakpoints. In contrast, for the DAG-Guided Active Scheduling, we injected a calculated delay (specifically $\Delta t = 1$ s, derived from our sensitivity analysis) directly into the race window identified by the DAG.

As shown in Figure 6, the Random Fuzzing baseline achieved a low reproduction rate of 12%, confirming the “Heisenbug” nature of the vulnerability—it is elusive and statistically rare. In stark contrast, our DAG-Guided approach achieved a 96% reproduction rate in this controlled fault injection test, confirming that theoretical race conditions can be deterministically verified.

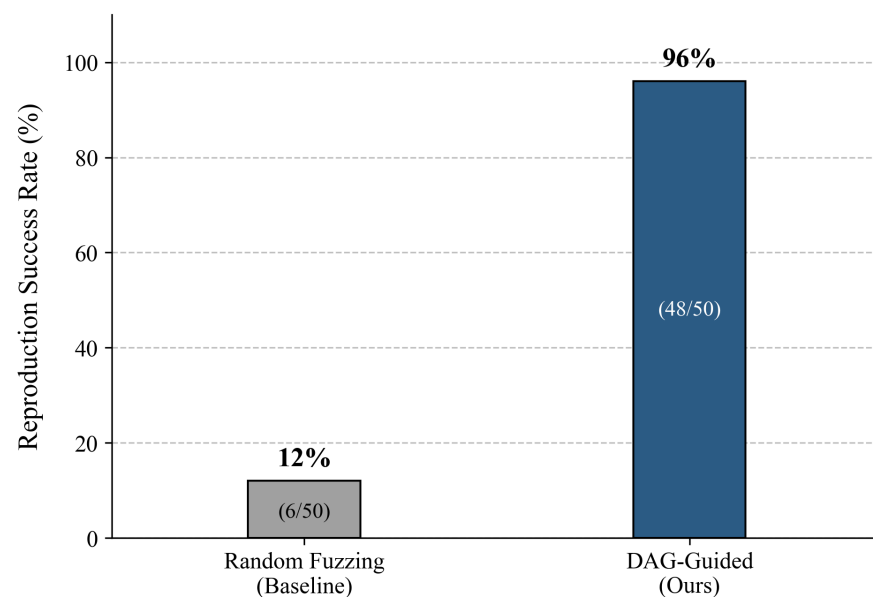


Figure 6. Stability test results under synthetic fault injection scenarios. Our deterministic approach transforms a transient “Heisenbug” into a consistently reproducible flaw.

4.6.2. Robustness Under Simulated Network Jitter

To evaluate robustness without a physical cluster, we utilized the Linux Traffic Control (tc) tool to inject artificial latency ($50 \text{ ms} \pm 10 \text{ ms}$ jitter) into the loopback interface of our All-In-One deployment. Under these noisy conditions, the detection rate of the Random/Passive baseline dropped to near zero ($<2\%$) as the injected noise masked the race signals. However, our Active Scheduling maintained a detection rate of over 85%. This confirms that by enforcing a dominant delay (e.g., 1000 ms) that significantly exceeds environmental jitter, our approach effectively “overpowers” network noise, ensuring robust detection even in unstable environments.

4.7. Limitations and Future Work

While our framework effectively detects intra-component race conditions, two primary limitations remain. First, regarding cross-subsystem visibility, our current DAG modeling focuses on single-service dependencies, as highlighted by the undetected case HBase-10257. Consequently, race conditions spanning multiple distinct subsystems (e.g., HBase interacting with Zookeeper) may escape detection if the causal link is not explicitly logged. While empirical studies (e.g., TaxDC) suggest that cross-subsystem races represent

a smaller fraction of total concurrency bugs, they remain a blind spot. Future work will extend the framework with Federated DAGs to capture these global dependencies. Second, concerning deployment scale, although our experiments on an All-In-One OpenStack deployment ensured strict experimental control, this setup may not fully replicate the massive concurrency pressure characteristic of a multi-node production cluster. Future work will focus on integrating Distributed Tracing (e.g., OpenTelemetry) to construct global dependency graphs, extending our active fuzzing capabilities to cover cross-service and multi-node concurrency scenarios.

5. Related Work

5.1. Cloud Service API Testing

Cloud services, predominantly accessed via RESTful APIs (e.g., AWS, Azure), rely on specifications like OpenAPI [23] to define their interfaces. Automated API testing has evolved from simple schema validation to sophisticated stateful interaction testing. RESTler [24] pioneered stateful REST API fuzzing by inferring producer-consumer dependencies from OpenAPI specs to generate valid request sequences. Extensions like Pythia [25] improved coverage by incorporating mutation-based learning, while Godefroid et al. [26] introduced differential regression testing to detect logic bugs across API versions.

However, existing API testing tools face significant limitations in detecting concurrency vulnerabilities:

- **Operation Scope:** Tools like RESTler primarily focus on CRUD (Create, Read, Update, Delete) operations [27]. They struggle with complex service logic that requires specific pre-provisioned states (e.g., attaching a volume to a specific VM state).
- **Deep State Reachability:** Approaches like MINER [28] and RESTLess [29] employ neural networks and LLMs to generate valid long sequences. While effective for functional correctness, they lack the temporal granularity required to trigger race conditions. They generate sequences that are semantically valid but temporally loose, failing to create the precise, microsecond-level interleavings needed to expose TOCTOU vulnerabilities.

Unlike these black-box sequence generators, our framework leverages DAG-based structural modeling to explicitly map the causal dependencies of API operations. This allows us to move beyond generating valid sequences to actively scheduling conflicting sequences with precise timing control.

5.2. Evolution of Fuzzing Techniques

Fuzzing has matured from random input generation to coverage-guided and structure-aware methodologies.

- **Grey-box Fuzzing:** Tools like AFL [14] and AFLFast [30] use lightweight instrumentation to guide input mutation towards unexplored code paths. AFLGo [31] further directs fuzzing towards specific target sites (e.g., changed code). However, these tools operate at the binary or file level and are ill-suited for the distributed, message-passing nature of cloud control planes.
- **Seed Generation Strategy:** Advanced seed generators like Zest [32] (structural fuzzing) and Montage [33] (neural language models) ensure syntactic validity [34]. While they reduce the search space for functional bugs, they do not account for the non-deterministic timing dimension inherent in distributed systems.
- **Race-Directed Fuzzing:** Recent works like AnKou [35] and GreyOne [36] use taint analysis to prioritize seeds affecting data flow dependencies. However, they typically target monolithic kernels or binaries.

We propose a Time-Aware Active Fuzzing strategy tailored for distributed systems. Instead of merely mutating data inputs, our fuzzer mutates scheduling constraints (e.g., injecting delays). By integrating the DAG model, we transform the fuzzing objective from maximizing code coverage to maximizing race pair coverage, enabling efficient discovery of concurrency bugs in cloud environments.

5.3. Advanced Race Detection Techniques

Beyond fuzzing, specialized techniques for race condition detection have emerged, generally categorized into SMT-based and Hybrid approaches.

5.3.1. SMT-Based Constraint Solving

Approaches like Spider [37] and RaceInjector [38] translate the concurrency scheduling problem into a Satisfiability Modulo Theories (SMT) formula. By exploring all mathematically feasible schedules, these tools theoretically achieve maximal coverage. However, the computational complexity of constraint solving leads to prohibitive runtime overhead (often exceeding 40%, as shown in our evaluation), making them impractical for continuous auditing of large-scale cloud systems.

5.3.2. Hybrid Static–Dynamic Analysis

To mitigate overhead, hybrid methods like SDILP [39] and IRHunter [40] utilize static analysis to identify high-risk code regions (e.g., locksets), narrowing the scope for dynamic verification. While efficient, static analysis in dynamic languages (like Python in OpenStack) suffers from high false positives due to the difficulty of alias analysis and dynamic dispatch. Furthermore, these tools often miss “logical races” where no memory corruption occurs, but the business logic is violated (e.g., quota bypass).

Our framework strikes an effective balance by combining Static DAG Extraction with Dynamic Active Scheduling. Unlike SMT methods, we do not exhaustively solve schedules but use the DAG to heuristically guide the scheduler towards high-probability race windows. This yields a detection rate comparable to formal methods (Recall 0.67) while maintaining the low overhead characteristic of fuzzing (11.5%), addressing the scalability gap in current research.

6. Conclusions

In this paper, we presented a systematic framework for securing distributed cloud infrastructure against non-deterministic concurrency attacks. Addressing the limitations of traditional passive testing, we introduced a novel DAG-Guided Active Fuzzing approach that combines static structural modeling with dynamic schedule enforcement.

By rigorously modeling the causal dependencies of API operations and actively injecting microsecond-level delays into critical execution windows, our system successfully transforms theoretical Time-of-Check to Time-of-Use (TOCTOU) risks into reproducible vulnerabilities. Extensive evaluations on the TaxDC benchmark and six industrial-grade distributed systems (including OpenStack, Hadoop, and Flink) demonstrated the robustness of our approach. We successfully identified 22 confirmed race conditions—comprising both order and atomicity violations—achieving a detection rate of 68.8%, which statistically outperforms baseline fuzzing techniques.

Our research highlights that the “Heisenbug” nature of distributed race conditions can be effectively tamed through deterministic active scheduling. This work not only provides a practical toolchain for cloud security auditing but also establishes a theoretical foundation for integrating time-aware fuzzing into the DevSecOps lifecycle of large-scale distributed systems. Future work will focus on extending the DAG modeling to capture global, cross-subsystem dependencies to address the challenge of multi-component race conditions.

Author Contributions: Conceptualization, H.Z. and D.Z.; methodology, Y.W.; software, H.Z.; validation, H.Z.; formal analysis, Z.L.; investigation, H.Z.; resources, D.Z.; data curation, H.Z.; writing—original draft preparation, H.Z.; writing—review and editing, Z.L.; visualization, H.Z.; supervision, Y.W.; project administration, Z.L.; funding acquisition, D.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This work is supported by the National Natural Science Foundation of China under Grant No. U2336203.

Data Availability Statement: The original data presented in this study are included in the article. Further inquiries can be directed to the author.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Cowan, C.; Beattie, S.; Wright, C.; Kroah-Hartman, G. {RaceGuard}: Kernel Protection From Temporary File Race Vulnerabilities. In Proceedings of the 10th USENIX Security Symposium (USENIX Security 01), Washington, DC, USA, 13–17 August 2001.
2. Loi, F.; Pisu, L.; Regano, L.; Maiorca, D.; Giacinto, G. Race against time: Investigating the factors that influence web race condition exploits. *Comput. Secur.* **2026**, *160*, 104740.
3. Cotroneo, D.; De Simone, L.; Liguori, P.; Natella, R.; Bidokhti, N. How bad can a bug get? An empirical analysis of software failures in the openstack cloud computing platform. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Tallinn, Estonia, 26–30 August 2019; pp. 200–211.
4. Musavi, P.; Adams, B.; Khomh, F. Experience report: An empirical study of API failures in OpenStack cloud environments. In Proceedings of the 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE), Ottawa, ON, Canada, 23–27 October 2016; pp. 424–434.
5. Musuvathi, M.; Qadeer, S.; Ball, T.; Basler, G.; Nainar, P.A.; Neamtiu, I. Finding and Reproducing Heisenbugs in Concurrent Programs. In Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, San Diego, CA, USA, 8–10 December 2008; Volume 8.
6. Liang, H.; Pei, X.; Jia, X.; Shen, W.; Zhang, J. Fuzzing: State of the art. *IEEE Trans. Reliab.* **2018**, *67*, 1199–1218.
7. Zeng, Q.; Kavousi, M.; Luo, Y.; Jin, L.; Chen, Y. Full-stack vulnerability analysis of the cloud-native platform. *Comput. Secur.* **2023**, *129*, 103173.
8. Leesatapornwongsa, T.; Lukman, J.F.; Lu, S.; Gunawi, H.S. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, Atlanta, GA, USA, 2–6 April 2016; pp. 517–530.
9. Lu, J.; Li, F.; Li, L.; Feng, X. Clouddraid: Hunting concurrency bugs in the cloud via log-mining. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Lake Buena Vista, FL, USA, 4–9 November 2018; pp. 3–14.
10. Xu, L.; Huang, J.; Hong, S.; Zhang, J.; Gu, G. Attacking the brain: Races in the SDN control plane. In Proceedings of the 26th USENIX Security Symposium (USENIX Security), Vancouver, BC, Canada, 16–18 August 2017; pp. 451–468.
11. Tang, H.; Wu, G.; Wei, J.; Zhong, H. Generating test cases to expose concurrency bugs in android applications. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Singapore, 3–7 September 2016; pp. 648–653.
12. Li, X.; Pan, D.; Wang, Y.; Ruiz, R. Scheduling multi-tenant cloud workflow tasks with resource reliability. *Sci. China Inf. Sci.* **2022**, *65*, 192106.
13. Gu, Y.; Mellor-Crummey, J. Dynamic data race detection for OpenMP programs. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, 11–16 November 2018; pp. 767–778.
14. Gutmann, P. Fuzzing Code with AFL. *Login Usenix Mag.* **2016**, *41*, 11–14.
15. Fioraldi, A.; Maier, D.C.; Zhang, D.; Balzarotti, D. Libafl: A framework to build modular and reusable fuzzers. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, Los Angeles, CA, USA, 7–11 November 2022; pp. 1051–1065.
16. Hong, X.J.; Yang, H.S.; Kim, Y.H. Performance analysis of RESTful API and RabbitMQ for microservice web application. In Proceedings of the 2018 International Conference on Information and Communication Technology Convergence (ICTC), Jeju Island, Republic of Korea, 17–19 October 2018; pp. 257–259.
17. Aoudni, Y.; Donald, C.; Farouk, A.; Sahay, K.B.; Babu, D.V.; Tripathi, V.; Dhabliya, D. Cloud security based attack detection using transductive learning integrated with Hidden Markov Model. *Pattern Recognit. Lett.* **2022**, *157*, 16–26.

18. Serebryany, K.; Iskhodzhanov, T. ThreadSanitizer: Data race detection in practice. In Proceedings of the Workshop on Binary Instrumentation and Applications, New York, NY, USA, 12 December 2009; pp. 62–71.
19. Cai, Y.; Chan, W.K. MagicFuzzer: Scalable deadlock detection for large-scale applications. In Proceedings of the 2012 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2–9 June 2012; pp. 606–616.
20. Lin, Y.; Kulkarni, S.S. Automatic repair for multi-threaded programs with deadlock/livelock using maximum satisfiability. In Proceedings of the 2014 International Symposium on Software Testing and Analysis, San Jose, CA, USA, 21–25 July 2014; pp. 237–247.
21. Han, X.; Schooley, R.; Mackenzie, D.; David, O.; Lloyd, W.J. Characterizing public cloud resource contention to support virtual machine co-residency prediction. In Proceedings of the 2020 IEEE International Conference on Cloud Engineering (IC2E), Sydney, Australia, 21–24 April 2020; pp. 162–172.
22. Kumar, V.A.; Das, D.; Senior Member IEEE. Data sequence signal manipulation in multipath TCP (MPTCP): The vulnerability, attack and its detection. *Comput. Secur.* **2021**, *103*, 102180.
23. Tzavaras, A.; Mainas, N.; Petrakis, E.G. OpenAPI framework for the Web of Things. *Internet Things* **2023**, *21*, 100675.
24. Atlidakis, V.; Godefroid, P.; Polishchuk, M. Restler: Stateful rest api fuzzing. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 748–758.
25. Atlidakis, V.; Geambasu, R.; Godefroid, P.; Polishchuk, M.; Ray, B. Pythia: Grammar-based fuzzing of rest apis with coverage-guided feedback and learning-based mutations. *arXiv* **2020**, arXiv:2005.11498.
26. Godefroid, P.; Lehmann, D.; Polishchuk, M. Differential regression testing for REST APIs. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual, 18–22 July 2020; pp. 312–323.
27. Du, W.; Li, J.; Wang, Y.; Chen, L.; Zhao, R.; Zhu, J.; Han, Z.; Wang, Y.; Xue, Z. Vulnerability-oriented testing for restful apis. In Proceedings of the 33rd USENIX Security Symposium (USENIX Security 24), Philadelphia, PA, USA, 14–16 August 2024; USENIX Association: Berkeley, CA, USA, 2024; pp. 739–755.
28. Lyu, C.; Xu, J.; Ji, S.; Zhang, X.; Wang, Q.; Zhao, B.; Pan, G.; Cao, W.; Chen, P.; Beyah, R. MINER: A Hybrid Data-Driven Approach for REST API Fuzzing. In Proceedings of the 32nd USENIX Security Symposium (USENIX Security), Anaheim, CA, USA, 9–11 August 2023; pp. 4517–4534.
29. Zheng, T.; Shao, J.; Dai, J.; Jiang, S.; Chen, X.; Shen, C. RESTLess: Enhancing State-of-the-Art REST API Fuzzing With LLMs in Cloud Service Computing. *IEEE Trans. Serv. Comput.* **2024**, *17*, 4225–4238.
30. Böhme, M.; Pham, V.T.; Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 1032–1043.
31. Lee, G.; Shim, W.; Lee, B. Constraint-guided directed greybox fuzzing. In Proceedings of the 30th USENIX Security Symposium (USENIX Security 21), Virtual, 11–13 August 2021; pp. 3559–3576.
32. Padhye, R.; Lemieux, C.; Sen, K.; Papadakis, M.; Le Traon, Y. Semantic fuzzing with zest. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, Beijing, China, 15–19 July 2019; pp. 329–340.
33. Lee, S.; Han, H.; Cha, S.K.; Son, S. Montage: A neural network language Model-Guided JavaScript engine fuzzer. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), Boston, MA, USA, 12–14 August 2020; pp. 2613–2630.
34. Reddy, S.; Lemieux, C.; Padhye, R.; Sen, K. Quickly generating diverse valid test inputs with reinforcement learning. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Seoul, Republic of Korea, 5–11 October 2020; pp. 1410–1421.
35. Manès, V.J.; Kim, S.; Cha, S.K. Ankou: Guiding grey-box fuzzing towards combinatorial difference. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Seoul, Republic of Korea, 5–11 October 2020; pp. 1024–1036.
36. Gan, S.; Zhang, C.; Chen, P.; Zhao, B.; Qin, X.; Wu, D.; Chen, Z. GREYONE: Data flow sensitive fuzzing. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), Boston, MA, USA, 12–14 August; pp. 2577–2594.
37. Pereira, J.C.; Machado, N.; Sousa Pinto, J. Testing for race conditions in distributed systems via SMT solving. In Proceedings of the International Conference on Tests and Proofs, Bergen, Norway, 22–23 June 2020; Springer: Cham, Switzerland, 2020; pp. 122–140.
38. Wang, M.; Srikant, S.; Samak, M.; O'Reilly, U.M. RaceInjector: Injecting Races to Evaluate and Learn Dynamic Race Detection Algorithms. In Proceedings of the 12th ACM SIGPLAN International Workshop on the State of the Art in Program Analysis, Orlando, FL, USA, 17 June 2023; Association for Computing Machinery: New York, NY, USA, 2023; pp. 63–70.
39. Bai, J.J.; Chen, Q.L.; Jiang, Z.M.; Lawall, J.; Hu, S.M. Hybrid static-dynamic analysis of data races caused by inconsistent locking discipline in device drivers. *IEEE Trans. Softw. Eng.* **2021**, *48*, 5120–5135.
40. Xin, G.; Xu, G.; Zhang, Y.; Wen, C.; Zhang, C.; Xie, X.; Xiong, N.N.; Liu, S.; Gao, P. IRHunter: Universal Detection of Instruction Reordering Vulnerabilities for Enhanced Concurrency in Distributed and Parallel Systems. *IEEE Trans. Parallel Distrib. Syst.* **2025**, *36*, 1220–1236.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.