USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Themis: Detecting Distributed Concurrency Bugs through RPC-Driven Race-Directed Test Generation and Fuzzing

Hongchen Cao and Jingzhu He, *ShanghaiTech University;*
Ting Dai, *InsightFinder AI;* Guoliang Jin, *North Carolina State University*

https://www.usenix.org/conference/nsdi26/presentation/cao

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

Themis: Detecting Distributed Concurrency Bugs through RPC-Driven Race-Directed Test Generation and Fuzzing

Hongchen Cao[†], Jingzhu He[†*], Ting Dai[‡], Guoliang Jin[◇]
[†]*ShanghaiTech University* [‡]*InsightFinder AI* [◇]*North Carolina State University*

Abstract

Distributed concurrency bugs occur when concurrent execution flows, at least one of which is triggered by inter-node communication such as remote procedure calls (RPCs), access the same shared variable or object in conflicting ways, causing incorrect behavior under certain interleavings. Existing work for detecting distributed concurrency bugs focuses on dynamic approaches, thus suffering from limited coverage. This paper proposes Themis, a novel approach that uses static analysis to detect potential races, applies LLM-based test generation, and employs directed fuzzing to refine input parameters for detecting distributed concurrency bugs. We have implemented a prototype of Themis and evaluated it on eight real-world distributed systems. Themis detects 198 new violations corresponding to 52 new bugs.

1 Introduction

Distributed systems form the backbone of modern computing infrastructures, powering large-scale data-processing frameworks, cloud services, and microservice platforms. Despite their critical role, these systems are notoriously prone to software bugs. Of these, distributed concurrency bugs caused by improper timing of distributed operations are among the most troublesome. They often evade in-house testing and manifest in production, leading to severe outages such as system downtime and service unavailability [49]. For example, in October 2025, a distributed concurrency bug caused a 14-hour DynamoDB outage in one of Amazon's major US regions [10], affecting Netflix, Zoom, and T-Mobile [8], with estimated losses of $38-$581 million [9]. The outage was caused by an unexpected interleaving of two concurrent DNS enactors executing on distributed nodes, where a delayed enactor applied an older plan that broke the atomicity of a newer enactor, resulting in an empty DNS record. While fixing the bug, developers introduced additional tests to catch similar issues [7], indicating a gap in the existing test suite.

*Jingzhu He is the corresponding author.

Distributed concurrency bugs require that at least one of the execution flows performing the conflicting accesses be triggered by inter-node communication, which distinguishes them from local concurrency bugs. In this paper, we focus on distributed concurrency bugs involving remote procedure calls (RPCs), a widely used communication mechanism. Specifically, we target cases where an RPC-triggered execution flow accesses a shared object concurrently with another flow, resulting in conflicting operations that lead to incorrect behaviors.

Prior research on detecting distributed concurrency bugs adopts dynamic approaches that monitor executions to infer happens-before relations and manipulate interleavings of accesses to shared objects [59–61]. Unlike local concurrency bug detection [44, 53, 64, 75, 85, 86], detecting distributed concurrency bugs requires reasoning about inter-node communication and monitoring communication-related objects. However, dynamic approaches inherently suffer from limited coverage, relying on hand-crafted workloads or existing test suites. While existing tools reduce runtime overhead to be potentially deployed in production to expand coverage, their effectiveness depends on the monitored workload exercising the relevant code paths and shared-state accesses; otherwise, latent bugs may remain unobserved and thus undetected.

To detect distributed concurrency bugs with both broad coverage and high precision, we propose a framework that integrates static detection of RPC-induced races, LLM-based test generation, and directed fuzzing for parameter refinement and interleaving exploration for bug exposure. Starting with static analysis maximizes coverage but inherently yields false positives due to analysis imprecision. To prune false positives while triggering actual bugs, we incorporate a Large Language Model (LLM) to generatively construct test cases for the detected bugs. As the LLM often fails to generate parameter values that satisfy complex path conditions, we employ directed fuzzing to fine-tune input parameters and explore different interleavings to expose true, harmful bugs.

Themis is built on the insight that distributed concurrency bugs cannot be reliably detected or exposed by one-shot or unguided test generation using an LLM alone. Instead, they re-

quire iteratively transforming potential races into concrete failing executions under strong structural guidance. Accordingly, Themis adopts an "LLM + guidance + checking" paradigm, in which static analysis constrains generation toward target code paths and fuzzing verifies and refines the results to mitigate LLM's hallucinations.

A key challenge for applying such a static-analysis-guided framework in distributed systems is their inherently open-ended nature: detecting concurrency bugs requires reasoning about executions driven by test harnesses that invoke public interfaces, yet such harnesses are typically unavailable a priori. Most existing static race detection tools target closed systems [24, 56]. While Chord [68], which detects races in Java programs statically, considers open systems, it relies on statically synthesized harnesses, and it is designed to detect local races whose root causes differ fundamentally from those of distributed concurrency bugs. In our framework, we perform static race detection before harness construction by assuming concurrent execution of each pair of public interfaces, and generate concrete harnesses only for triggering and validating the detected races. While this assumption over-approximates the concurrency model and yields false positives, we prune them using the subsequent test generation and validation steps.

Proceeding without test harnesses requires a different static race detection workflow. In closed or open systems with a harness, a top-down analysis can be performed to enumerate concurrent execution flows, extract access sites, and analyze pairs of access sites to detect races, defined as two concurrent, unsynchronized accesses to the same shared memory location where at least one is a write. Without test harnesses, our static analysis is variable-centric, focusing on RPC-reachable shared variables and their access sites. Starting from an RPC client-server pair, the analysis traverses the RPC execution path to identify shared variables of interest and aggregates access sites for each variable, including those outside the original RPC path. These access sites allow the analysis to construct the conflicting execution flow, which would otherwise need to be derived from a test harness. To improve precision, the analysis only reports when the racy accesses can form atomicity violations and order violations with observable symptoms. This process identifies harmful distributed concurrency-bug candidates under the assumed concurrency model where any pair of public interfaces can be concurrent.

To validate potential bugs from the static detection step, Themis constructs a test harness for each of them, and the process involves two orthogonal tasks: call-sequence construction and parameter-value generation, where the former incorporates necessary calls to prepare the system state so that the public interfaces can execute without errors and follow the execution paths as in the detection results, and the latter generates parameter values that satisfy path constraints. To avoid the combinatorial explosion of searching for sequences and parameters simultaneously, we decouple these tasks: we use an LLM for structural call-sequence synthesis and coarse parameter seeding, leaving constraint satisfaction and fine-grained parameter exploration to directed fuzzing.

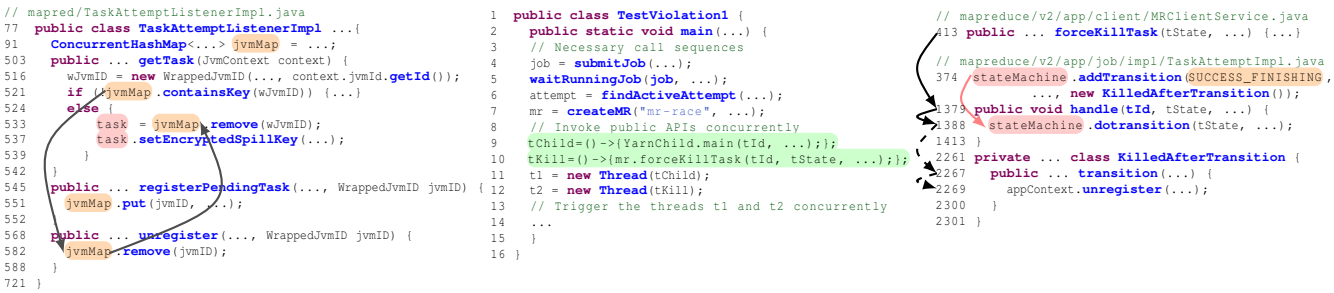
Prior work on synthesizing tests for concurrency bugs either reuses existing tests of the two racy functions [35, 73, 74], or generates unified sequence templates [37, 68, 71, 77]. However, triggering distributed concurrency bugs requires highly diverse, system-specific call sequences to prepare the necessary state before invoking public interfaces, a task that existing non-LLM techniques based on templates cannot readily support without substantial additional effort. To overcome this, we use an LLM to generate tests for statically detected bugs, guiding it to synthesize the required state-preparing call sequences and invoke public interfaces with coarse parameter values that execute without runtime errors.

To balance accuracy and efficiency, after generating the "basic" test cases that may have the necessary call sequences but not the parameter values satisfying the path constraints, Themis applies directed fuzzing, a technique that adjusts parameters to drive execution toward targeted statements, to refine input parameters to reach the memory accesses of the detected bugs. Once these accesses are reached, we record the current state (seed pool and parameter values) and manipulate interleavings to expose the bugs. Even then, distributed concurrency bugs may not immediately manifest due to unsatisfied failure conditions. To handle this, we initialize a new fuzzing procedure that inherits the seed pool and starts from the current input, adjusting parameters further to expose observable symptoms.

1.1 A Motivating Example

Figure 1 shows MapReduce-7507, a new bug detected by our tool and subsequently confirmed by developers. Before detailing the bug, we clarify the two distinct client-server relationships involved in RPC-based systems. First, at the system level, external clients (e.g., users) invoke public interfaces exposed by the server (e.g., MapReduce). Second, at the internal level, handling a request often requires communication between nodes within the system; in this context, the node initiating the request is the RPC client, and the node handling it is the RPC server. In Figure 1, `getTask()` is an RPC server function invoked by the RPC client function `YarnChild.main()`. Notably, `YarnChild.main()` is also a public interface exposed to external clients.

The bug can manifest as an atomicity violation in two scenarios. First, RPC server functions `getTask()` and `unregister()` may execute concurrently on the RPC server when `YarnChild.main()` is invoked concurrently with another public interface `forceKillTask()`. In this scenario, the `remove` at line 582 in `unregister()` occurs between the `if` check at line 521 and the `remove` at line 533 in `getTask()`. Second, an external client may invoke `YarnChild.main()` in two threads, and the RPC server executes `getTask()` in separate threads concurrently. The two `getTask()` may interleave



(a) The simplified code containing two atomicity violations on `jvmMap`. → shows the buggy interleaving 521-582-533.

(b) The generated test case for violation 521-582-533. Lines 4–7 show the necessary call sequence to prepare system states.

(c) With LLM-generated coarse parameter values, execution reaches only lines 1379 and 1388 in the `TaskAttemptImpl` class.

Figure 1: Overview of MapReduce-7507, a newly detected and confirmed distributed concurrency bug.

such that the `remove` at line 533 in one thread occurs between the `if` check at line 521 and the `remove` at line 533 in the other thread. In both cases, the `remove` in `getTask()` returns `null`, causing the dereference at line 537 to raise `NullPointerException`. We next discuss how Themis detects this bug.

Our approach begins with static analysis. We first identify an RPC client–server pair, namely `YarnChild.main()` and `TaskAttemptListenerImpl.getTask()`. From this pair, we traverse upward through the RPC client's call chain and downward through the RPC server's call chain to extract RPC-reachable variables. The analysis identifies `jvmMap` on the RPC server side, accessed at lines 521 and 533 in `getTask()`. We then locate additional accesses, including accesses at line 551 in `registerPendingTask()` and line 582 in `unregister()`. Applying our static checking rules yields three candidate atomicity violations (521-533-533, 521-551-533, and 521-582-533) and two order violations (533-551 and 533-582). The two order violations are subsumed by the atomicity violations (521-551-533 and 521-582-533) because they have the same racy accesses and symptom statements, and they are not processed separately in subsequent stages. All violations match the missing retrieval check (MR) pattern, where an entry is returned at line 533 and then dereferenced without a null check at line 537.

For each detected violation, an LLM supplied with static analysis results generates a basic test case that prepares the system state and invokes public interfaces with coarse parameter values without triggering errors. Figure 1b shows the generated test for violation 521–582–533, which concurrently invokes the public interfaces `YarnChild.main()` and `forceKillTask()`. The call sequence first initializes the RPC server via job submission, which instantiates an instance of the `TaskAttemptListenerImpl` class. It then waits for the job to run and locates the active attempt to establish the program state needed to enter the `else` branch at line 524 of `getTask()` in Figure 1a. Finally, `createMR()` instantiates an `mr` object, which is used to invoke `forceKillTask()`. The generated call sequences span heterogeneous operations, such as job submission, runtime object instantiation, and multi-stage state preparation across multiple classes, but lack predefined invocation patterns, and are therefore difficult to enumerate in advance.

The basic test case generated by LLM, which sets `tState` to `ASSIGNED`, failing the path condition in `addTransition()` (line 374) and preventing `unregister()` from being invoked, and Figure 1c illustrates the call path. Directed fuzzing identifies `SUCCESS_FINISHING` as the required value, enabling execution to reach the racy accesses (lines 521, 533, and 582 in Figure 1a), after which we manipulate interleavings to expose the bug. Violations without observable symptoms after adjusting parameters and exploring interleavings are pruned, and violation 521-551-533 is pruned as a false positive.

1.2 Contributions

We design and implement Themis[1], a novel framework for detecting distributed concurrency bugs. As shown in Figure 2, Themis consists of three modules. First, it performs static race detection tailored to open systems by traversing RPC client–server call chains both upward and downward to extract RPC-reachable variables and identify their access sites. It then applies violation analysis rules, retaining only those reachable from public interfaces and matching specific error patterns. Second, Themis generates test harnesses for public interfaces with call sequences that establish the target program states, though parameter values may not initially satisfy all path conditions. Third, Themis applies parameter fuzzing to drive execution to racy accesses, manipulates interleavings and checks resulting behaviors, and then launches a follow-up fuzzing phase to satisfy error-triggering conditions and expose the targeted symptoms.

Overall, this paper makes the following contributions:

- We present Themis, a novel framework for detecting distributed concurrency bugs that integrates static race detection, LLM-based test harness generation, and parameter fuzzing with interleaving exploration to substantially improve detection coverage.

[1]Themis is the Greek goddess of order who enforces correctness in distributed concurrency chaos.

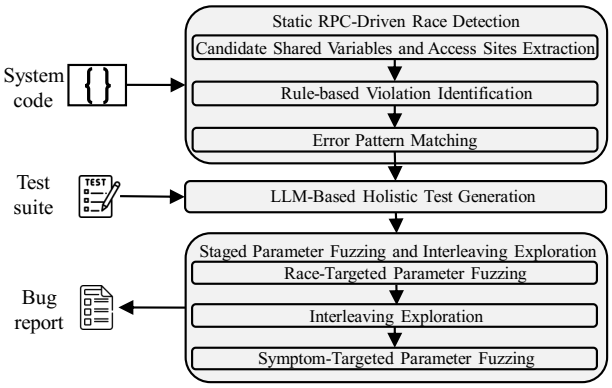


Figure 2: The Themis framework.

- We propose a static RPC-driven race detection that tracks RPC-reachable variables and applies violation-specific rules with error pattern matching. Guided by this analysis, we develop LLM-based harness generation to construct state-preparing call sequences, a distributed-system–aware directed fuzzing strategy to refine parameters, and interleaving manipulation to trigger bugs.
- We have implemented a prototype of Themis, publicly available at https://github.com/caohch-1/Themis. Evaluation on eight real-world distributed systems shows that Themis detects 204 violations (of which 198 are new) corresponding to 58 bugs (of which 52 are new). We have reported all the 52 newly discovered bugs to developers, and six have been confirmed as of publication.

2 Related Work

Distributed Concurrency Bug Detection and Fixing. Distributed concurrency bugs extend concurrency challenges to multi-node systems, where shared memory accesses are mediated through RPCs or message passing [59–61]. Existing detection tools, i.e., DCatch [59] and CloudRaid [60, 61], adopt dynamic approaches and suffer from coverage limitations. DFix [52] performs automatic patching using carefully-defined roll-backs and fast-forwards. Compared with prior work, Themis starts with static analysis followed by test generation and fuzzing, achieving higher detection coverage.

Dynamic Detection of Local Concurrency Bugs. Dynamic techniques for detecting concurrency bugs in single-process, multi-threaded programs have been extensively studied [38, 39, 44, 45, 53, 57, 64, 67, 75, 82–87]. Existing approaches either perform substantial analysis to identify suspicious racy accesses followed by targeted testing [39, 44, 53, 64, 75, 83, 85, 86], or rely on heuristic exploration of thread interleavings with little or no prior analysis [38, 45, 57, 67, 82, 84, 87]. However, they depend on hand-crafted workloads or existing test suites and suffer from coverage limitations.

Static Race Detection. Prior work on static race detection primarily targets local concurrency bugs in closed systems [24, 31, 42, 43, 56, 72, 79]. Chord [68] considers open systems but begins analysis from synthesized harnesses. In contrast, Themis targets open systems and follows an analysis-first pipeline, using static analysis to guide the construction of diverse and system-specific test harnesses for triggering distributed concurrency bugs.

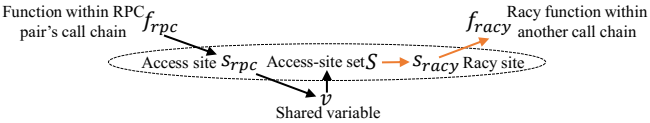


Figure 3: The static analysis workflow. → represents the workflow of Section 3.1, while → represents the workflow of Section 3.2.

Concurrency Test Generation. Prior work on automated concurrency test generation primarily targets unit-level multi-threaded tests for local races [71, 73, 74, 77, 78, 80]. These approaches do not generalize to distributed concurrency bugs, as they cannot construct the diverse, system-specific call sequences or orchestrate multi-component interactions required to establish the necessary system states.

Parameter Fuzzing. Parameter fuzzing includes coverage-guided [2–5, 33, 70] and directed approaches [25, 28, 48, 54, 55, 65, 69, 76]. Coverage-guided fuzzing maximizes code coverage, whereas directed fuzzing steers execution toward target locations. Prior work applies parameter fuzzing to concurrency bug detection by optimizing concurrency-related coverage metrics and injecting delays to expose bugs [27, 39]. Themis builds on directed fuzzing [25, 65] to guide exploration toward racy accesses and their failure symptom paths.

3 Static RPC-Driven Race Detection

In this paper, we focus on RPC-induced distributed concurrency bugs, where shared variables are concurrently accessed without proper synchronization, with at least one access occurring along an RPC call chain. These shared variables may reside on either the RPC server or the RPC client. In both cases, one access occurs along an RPC execution, either within an RPC server handler or mediated through RPC parameters or return values on the client, while the conflicting access may originate from another concurrent RPC or from a local call on the same side where the variable resides.

Distributed systems, by their open-systems nature, expose public interfaces that may be invoked concurrently by external clients once the appropriate system state is established. Our static analysis identifies pairs of such public interfaces and their corresponding call chains, with at least one being an RPC call chain. Because constructing test harnesses that establish the required system state often involves complex and non-trivial call sequences, Themis performs static analysis without relying on test harnesses. Instead, it conservatively assumes that any pair of public interfaces can be invoked concurrently, and defers test-harness generation to a subsequent step.

Figure 3 illustrates the workflow of Themis's static analysis. Starting from an RPC client-server pair, Themis traverses the call chain both upward and downward to identify RPC-reachable access sites (denoted as s_{rpc}) and extracts candidate shared variables (denoted as v). The upward search identifies client-side variables, while the downward search identifies server-side variables. It then gathers all access sites to each shared variable, forming an access-site set S, which also includes accesses outside RPC call chains to capture races containing at least one RPC-reachable access (Section 3.1). Themis applies violation-specific static rules to check whether a variable escapes to both the RPC (via access site s_{rpc}) and another thread via access site s_{racy} from S, detecting either *order violations*, where accesses occur out of the intended sequence, or *atomicity violations*, where accesses interleave, breaking the expected atomic execution (Section 3.2).

Themis prunes benign races by checking whether each detected violation can lead to an observable symptom. Only symptom-producing violations are retained, and for each, Themis outputs the corresponding call path, including the public interface pair, RPC client–server pair, racy access statements together with their enclosing functions, and potential symptom statement, to guide subsequent test generation (Section 4) and fuzzing (Section 5).

3.1 Candidate Shared Variables and Access Sites Extraction

Themis begins by extracting RPC client-server pairs to establish caller-callee relationships. It then traverses the RPC call chain both upward and downward to identify access sites and candidate shared variables on the client and server sides. For each variable, Themis constructs an access-site set that includes sites both within and outside the RPC chain. This set enables race detection by identifying conflicting accesses to shared variables across different call chains.

RPC Client–Server Pair Extraction. Correct RPC communication requires both protocol alignment and compatible endpoint addresses. Themis uses RPCBridge [30] to automatically extract candidate RPC client–server pairs that implement the same protocol. Since RPCBridge does not analyze endpoint addresses, Themis performs a configurable-address analysis: it tracks configuration values to RPC endpoints and identifies address-related keys. Pairs whose client and server rely on disjoint configurable addresses are pruned. Pairs sharing an address but differing in ports may remain as false positives and are filtered later during fuzzing (Section 5.1).

Access-Site Extraction. Themis performs reachability analysis by traversing both upward through the RPC client's call chain and downward through the RPC server's call chain to identify relevant memory access sites (denoted as s_{rpc}) to local, instance, static, and parameter variables. At this stage, Themis considers all access sites without checking whether they are reachable from a public interface, deferring public-interface identification to Section 3.2. The downward traversal examines the server function and its callees to locate access sites within the RPC server. On the client side, since RPC behaves like a local procedure call (LPC), variable accesses are mediated through parameters or return values. Thus, Themis traverses the RPC client's callers via the inter-procedural control-flow graph (ICFG) to identify access sites that propagate into RPC parameters or consume RPC return values.

Variable Extraction from Access Sites. For each access site, Themis extracts the corresponding accessible variables and classifies the access as either a read or a write. For parameter variables and receiver objects, Themis employs points-to analysis [50] to identify the referenced variables. Each referenced variable is then similarly classified as a read or a write. If a referenced variable is passed to a library or third-party call, Themis classifies the access as a read to avoid missing potential data races. Further details on access sites and read–write access classification are provided in Appendix A.1.

Candidate Shared Variable Filtering. Once variables are extracted, Themis filters them to retain only those that may be shared and can lead to distributed concurrency bugs. As the execution models of RPC servers and clients differ, Themis applies distinct filtering strategies on the two sides.

On the RPC server side, the filtering strategy is guided by the server-side RPC execution model, where the RPC server is a long-lived instance that concurrently handles multiple client requests via worker threads. As a result, instance fields of the RPC server are shared across threads and constitute a primary source of distributed concurrency bugs. Themis therefore tracks server-instance fields, static fields, and IO/NIO objects within the RPC server class, and conservatively retains static and IO/NIO variables in inter-class callees. We do not track inter-class instance fields, as objects instantiated at different sites typically represent distinct instances; an examination of the evaluated systems revealed no cases of such inter-class instance-field sharing. On the RPC client side, all local, instance, and static variables are retained, as client-side execution does not introduce the same implicit cross-request sharing as on the server side.

Per-Variable Access-Site Set Construction. For each candidate shared variable v retained after filtering, Themis constructs an access-site set S consisting of all accesses that may come from another call chain different from the specified RPC call chain and conflict with the RPC-reachable access s_{rpc}. To build this set, Themis examines all the access sites within the variable's declaration scope and applies points-to analysis [50] to retain in S only those accesses that reference the same shared variable, classifying each as a read or a write. Specifically, the scope is restricted to the declaring class for instance variables, the declaring method for local variables, and extended to the entire codebase for static variables, which are globally visible. For IO/NIO objects, Themis performs additional path-based analysis to identify underlying resources and determine whether different objects may

Table 1: Notations for violation analysis rules in Table 2.

Symbol	Meaning
$\mathcal{S}_v$	Statements $\mathcal{S}_v$ are access sites to variable v.
$\overset{\vee}{\sim}(f,\mathcal{S}_v)$	Within the variable v's declaration scope, function f is the closest enclosing function that reaches statements $\mathcal{S}_v$ through ICFG on the same control-flow consecutively.
$\overset{\Vert}{\sim}(f,\mathcal{S}_v,\mathcal{S}_v')$	Within the variable v's declaration scope, function f is the closest enclosing function that reaches statements $\mathcal{S}_v$ and $\mathcal{S}_v'$ through ICFG on the same control-flow asynchronously.
$\mathbb{W}(\mathcal{S}_v)$	At least one statement in $\mathcal{S}_v$ performs write operation.
$\mathbb{A}(\mathcal{S}_v,\mathcal{S}_v')$	Consecutive statements $\mathcal{S}_v$ interleave with another statement $\mathcal{S}_v'$ in one of four patterns including RWR, WWR, WRW, and RWW.
$viol\langle\mathcal{F},\mathcal{S}_v,T\rangle$	Detected violation tuple with enclosing functions $\mathcal{F}$, access statements $\mathcal{S}_v$, and type T (Order/Atomicity).

Table 2: Violation analysis rules. In Rules ❶ and ❷, f_1 and f_2 must be reachable through different call chains. $\mathcal{S}_v=\{s_1,s_2\}$ in Rules ❶, ❸, and ❹. $\mathcal{S}_v=\{s_1,s_2\}$ and $\mathcal{S}_v'=\{s_3\}$ in Rules ❷ and ❺. Starting from the RPC-reachable access site $s_{rpc}\in\{s_1,s_2,s_3\}$, at least one of f_1 and f_2 must be part of the RPC call chain.

Violation Analysis Rules	Violation Tuple
❶ $\overset{\vee}{\sim}(f_1,\{s_1\})\wedge\overset{\vee}{\sim}(f_2,\{s_2\})\wedge\mathbb{W}(\mathcal{S}_v)$	$viol\langle\{f_1,f_2\},\mathcal{S}_v,Order\rangle$
❷ $\overset{\vee}{\sim}(f_1,\mathcal{S}_v)\wedge\overset{\vee}{\sim}(f_2,\mathcal{S}_v')\wedge\mathbb{A}(\mathcal{S}_v,\mathcal{S}_v')$	$viol\langle\{f_1,f_2\},\{s_1,s_2,s_3\},Atomicity\rangle$
❸ $\overset{\vee}{\sim}(f_1,\mathcal{S}_v)\wedge(\mathbb{A}(\mathcal{S}_v,\{s_1\})\vee\mathbb{A}(\mathcal{S}_v,\{s_2\}))$	$viol\langle\{f_1\},\mathcal{S}_v,Atomicity\rangle$
❹ $\overset{\Vert}{\sim}(f_1,\{s_1\},\{s_2\})\wedge\mathbb{W}(\mathcal{S}_v)$	$viol\langle\{f_1\},\mathcal{S}_v,Order\rangle$
❺ $\overset{\vee}{\sim}(f_1,\mathcal{S}_v)\wedge\overset{\Vert}{\sim}(f_1,\mathcal{S}_v,\mathcal{S}_v')\wedge\mathbb{A}(\mathcal{S}_v,\mathcal{S}_v')$	$viol\langle\{f_1\},\{s_1,s_2,s_3\},Atomicity\rangle$

refer to the same resource. File paths are treated as implicit references, analogous to pointer dereferences, and the analysis scope therefore includes all IO/NIO access sites across the codebase. Because paths may be dynamically constructed via function calls or string concatenation, Themis leverages StringAnalyzer [1,29], a static string solver that approximates possible runtime file paths, enabling identification of potential aliasing among IO/NIO objects.

3.2 Rule-Based Violation Identification

Themis detects distributed concurrency violations by reasoning over the shared-variable access sites extracted in the previous step. It identifies a race when one access to a variable, derived from an RPC call chain, conflicts with another access to the same variable along a different call chain, where the latter access belongs to the constructed access-site set. To identify concurrency violations from racy accesses, Themis applies the predefined rules in Table 2, which capture common atomicity and order violation patterns under RPC execution. Themis further extracts the public interfaces of the two call chains to confirm that the variable can escape to two different threads, followed by the unlocked-violation tuple analysis to prune atomicity violations that are properly synchronized.

Themis produces violation tuples $\langle\mathcal{F},\mathcal{S}_v,T\rangle$, where $\mathcal{F}$ represents the enclosing functions, $\mathcal{S}_v$ is the set of conflicting access sites to the shared variable v, and T denotes the violation type (atomicity or order violation). Themis also includes the corresponding public interfaces that can invoke each violation, which are used to construct test harnesses in Section 4.

Violation Analysis Rules. Starting from an RPC-reachable variable v and an access site s_{rpc} with the enclosing function f_{rpc}, Themis searches for other racing accesses to the same variable along a different call chain and applies the static rules in Table 2 to identify either order or atomicity violations, as outlined below. Details on the completeness proof of the rules are provided in Appendix A.2.

- Rule ❶ (Order violation across different functions): if two functions f_1 and f_2 respectively reach two access sites $\{s_1,s_2\}$ to the same variable v, and at least one access is a write, then Themis reports a potential order violation, since the two accesses may interleave in a nondeterministic order.
- Rule ❷ (Atomicity violation across different functions): if f_1 consecutively accesses the same variable v at $\{s_1,s_2\}$, while another function f_2 reaches an additional access s_3, and the three accesses can form one of the four buggy interleaving patterns (RWR, WWR, WRW, RWW), Themis reports a potential atomicity violation. The motivating example in Figure 1a illustrates such a violation between `unregister()` and `getTask()`.
- Rule ❸ (Self-atomicity violation within one function): if a function f_1 consecutively accesses the same variable at $\{s_1,s_2\}$, concurrent executions of f_1 may interleave with each other. If such interleavings match one of the four buggy patterns, Themis reports a potential atomicity violation.
- Rule ❹ (Order violation via asynchronous execution): if a function f_1 concurrently accesses the same variable v at $\{s_1,s_2\}$ through asynchronous mechanisms (e.g., `Callable`, `Runnable`, `Thread`, or `Future`), denoted as $\overset{\Vert}{\sim}(f_1,\{s_1,s_2\})$, and at least one access is a write, Themis reports a potential order violation.
- Rule ❺ (Atomicity violation via asynchronous execution): if a function f_1 consecutively accesses a variable at s_1,s_2 while also concurrently accessing the same variable at s_3 through asynchronous mechanisms, and the interleaving matches one of the four buggy patterns, Themis reports a potential atomicity violation.

Server and Client Distinctions. Rules ❶–❸ apply to both server and client code. The reason is that, on the server side, RPC frameworks execute each call concurrently in a separate thread, while on the client side, open systems allow diverse user-initiated invocations [34], which ensures that distinct functions can execute concurrently. In contrast, Rules ❹ and ❺ are specific to the client side, since on the server side, asynchronous mechanisms only create local multithreading independent of RPC clients, and such races are therefore outside the scope of distributed concurrency bugs. Moreover, variable scope differs between the two sides: local variables are considered only on the client side and appear exclusively in Rules ❹ and ❺.

Multi-Variable Case Analysis. Although some cases involve multiple variables within the same functions, true multi-variable races arise only when these variables are correlated,

i.e., when they must remain consistent or jointly satisfy an invariant [62]. Themis detects no such correlations, and our single-variable focus is sufficient.

Public Interface Pair Extraction. For each violation, Themis extracts a pair of public interfaces corresponding to the RPC and conflicting chains. If a public interface is absent for either call chain, the violation is discarded as non-escaping and untriggerable. For violations detected by Rules ❶–❸, it extracts the public interface pairs invoking the enclosing functions f_1 and f_2. For server-side violations, Themis extracts public interfaces upward along the RPC client's call chain for the specified RPC access. If the conflicting function is on another RPC call chain, it extracts public interfaces similarly on the RPC client side, forming a race between two concurrent RPC calls; otherwise, it extracts public callers within the racy variable's declaration scope, identifying a race between an RPC call and a local call. For client-side violations, Themis ensures conflicting accesses by extracting public interfaces upward along both call chains within the racy variable's declaration scope. For violations detected by Rules ❹ and ❺, where only one enclosing function is detected and concurrency is ensured, Themis extracts public interfaces invoking the function along the call graph.

Synchronization Analysis. For atomicity violations, we further prune the candidate cases by checking whether the function containing the two accesses is declared with the `synchronized` keyword, or whether both accesses are enclosed within a `synchronized` block. Identifying application-defined synchronizations (e.g., locks protecting critical sections and condition variables enforcing execution order [40, 41, 52, 58, 63]) requires domain knowledge and increases the complexity of static analysis. Ignoring application-specific locks at this stage can introduce false positives. To balance accuracy and efficiency, we defer eliminating such cases involving synchronization until the testing phase.

3.3 Error Pattern Matching

Concurrent accesses to the same object are pervasive in distributed systems, and many are intentional and benign by design [59, 64, 85], while true harmful bugs typically exhibit observable symptoms. We distinguish true concurrency bugs from benign races by examining whether the latent execution leads to *explicit* symptoms, which are developer-defined failures like fatal logs, or *implicit* symptoms, which are unhandled runtime errors like null pointer dereferences. Building on prior work [59–61, 64, 74, 85, 86] and our observed violations, we define seven patterns to capture explicit and implicit symptoms and describe them below. A violation tuple can lead to multiple symptoms, each counted as a unique violation. Further details of these patterns are provided in Appendix A.3.

Explicit Symptom Detection. For each identified violation, Themis checks whether the erroneous internal state after racy accesses propagates through control and data dependencies to explicit failure statements, including (1) uncaught exceptions, (2) severe error logs (e.g., `Log.error`), or (3) system aborts (e.g., `System.exit`), consistent with prior studies [59–61]. We name such violations explicit order violations (**EO**) and explicit atomicity violations (**EA**). Themis checks control and data dependencies between racy accesses and the failure statements on top of the call graph enhanced with RPC connections, enabling it to detect bugs that propagate erroneous internal state via RPCs. Prior studies show that error propagation distance is typically short [59, 85]. To balance efficiency and accuracy, we bound the call-path hops from a racy access site to an observable symptom to three, which was sufficient to capture all explicit symptoms in our evaluation.

Implicit Symptom Detection. Themis focuses on two common types of implicit symptoms: null pointer and out-of-bounds exceptions, which are among the most common sources of latent errors in distributed systems and account for a substantial portion of observed concurrency-related failures [51]. Themis includes five error patterns to capture violations likely to lead to these two implicit runtime exceptions. Among the five patterns, four of them do not require erroneous state propagation, and the symptom manifests on one of the racy accesses. Each violation is classified into a pattern based on its violation type, shared object type, and interleaving pattern. The five patterns are outlined below.

- **Null Pointer Dereference (NP):** Dereferencing an object after it is assigned to `null`.
- **Out-of-Bounds Access (OB):** An array-like object is concurrently accessed, causing one of the accesses to fall outside its valid bounds.
- **Uninitialized Read (UR):** For an order violation, the read happens before the write operation (instantiation).
- **Missing Retrieval Check (MR):** The read operation (retrieval) from a collection or array may return `null`. The bug occurs if the returned value is used directly without a null check. Bugs matching this pattern require error propagation from a racy access to a later dereference.
- **Thread-Unsafe Objects (TO):** Concurrent accesses to labeled thread-unsafe objects.

Overlapping Atomicity and Order Violations. The same racy accesses could constitute both an atomicity violation and an order violation. For example, related to the race between lines 533 and 582 in Figure 1, Themis reports both an atomicity violation (521-582-533) and an order violation (533-582). Themis considers an order violation subsumed by an atomicity violation if its racy accesses are a subset of the atomicity violation's and they share the same symptom statement. Subsumed violations are merged rather than processed separately in subsequent stages. This does not introduce false negatives because Themis exercises all relevant interleavings during the triggering phase; even if the root cause is truly an order violation, the bug will still be detected. Finally, we exclude subsumed order violations from our reported violation counts to avoid double-counting.

4 LLM-Based Holistic Test Generation

Given the execution paths (including the RPC pair, the violation tuple $\langle \mathcal{F}, \mathcal{S}_v, T \rangle$, the public interface set, and the symptom statement) for each potential violation, Themis incorporates the static analysis results to guide an LLM to generate a test case that first establishes the appropriate system state through necessary call sequences, and then invokes the public interfaces without runtime errors such as crashes or invalid invocations. This step focuses on producing a runnable test; however, the tests generated by the LLM in this step, even when provided with the violation tuple and symptom statement, rarely reach the racy access sites or trigger the symptom. These test cases serve as the basis for the next stage (Section 5), which refines parameters and explores interleavings.

Generation-Repair Workflow. For each violation, Themis follows a generation-repair workflow. It first generates an initial test using a prompt that incorporates the static-analysis results. Because the initial test often executes with runtime errors, the test is iteratively repaired using error messages and relevant code until a repair threshold is reached. If no test that executes without runtime errors is produced within the threshold, the workflow is restarted, leveraging the LLM's non-determinism to obtain a different initial test. After several unsuccessful cycles, existing test suites are incorporated into the generation prompt to leverage their call sequences, while the repair process and restarting strategy remain unchanged.

To enable effective test generation, Themis provides explicit guidance and narrows the scope through careful prompt design. Each prompt consists of two complementary components: instruction and context [26, 32, 81], which effectively guide the LLM and focus its attention. Detailed prompts are provided in Appendix A.4, and we summarize their design and usage below.

Using a generation prompt, Themis supplies the full execution paths from the public interfaces together with the RPC server class code, and instructs the LLM to initialize the RPC server, construct call sequences for triggering the execution paths, and invoke the public interfaces concurrently. We then execute the generated tests to ensure they run without runtime errors. Before execution, the system is manually booted to avoid failures caused by uninitialized components, and each test is then executed on the running system with the highest privilege level to prevent permission-related RPC failures.

If runtime errors occur, Themis repairs the test using a repair prompt that provides the current test, the execution error message, and the class code containing the error, and explicitly instructs the LLM to fix the issue. This repair process continues until the test executes without errors or the repair threshold (eight in our evaluation) is exceeded. To mitigate LLM hallucination, Themis performs multiple generation–repair cycles (five in our evaluation), leveraging regeneration to produce different initial test cases, thereby increasing the likelihood that at least one can be successfully repaired within the threshold.

If generation still fails after five cycles, Themis augments the generation prompt with existing test suites. Although these test suites rely on mocking mechanisms, they can provide valid call sequences and arguments. All test suites from classes containing the public interfaces are added as context, and Themis uses the same generation–repair workflow and workflow restarting strategy.

Public Interface Selection. A violation may be reachable through multiple public interfaces, all of which are identified by our static analysis. We prioritize public interfaces with the highest likelihood first. Once test construction succeeds for one interface, no further attempts are made for other interfaces associated with the same violation. Themis first tries the closest public interfaces, defined by the minimal number of call-graph hops to the enclosing functions, as they impose the fewest additional path conditions. It then considers top-level public interfaces (i.e., those without callers), which are closest to user-facing entry points (e.g., CLI APIs) and typically require shorter call sequences and simpler parameter values, making them easier for the LLM to exercise correctly. If neither succeeds, Themis progressively explores other public interfaces from closest to farthest in hop distance, increasing the likelihood of generating a successful test.

LLM-Generated Parameter Value Set. In addition to the single parameter value set used in the test case, Themis further instructs the LLM to generate multiple candidate parameter value sets that serve as seeds for directed fuzzing. Although LLM-generated parameter value sets often fail to reach racy access sites directly, they are generally stronger than conventional fuzzing seeds, providing values closer to valid ones and thus accelerating fuzzing.

Insufficient Call Sequence. Despite carefully guiding test generation with static analysis results and focused prompts, Themis may still produce tests that, while error-free at runtime, lack the necessary call sequences to establish the required system state for triggering targeted accesses. Directed fuzzing refines input parameters but cannot recover missing call sequences. As a result, such tests fail to expose violations, leading to five false negatives (Section 6.2.3).

5 Staged Parameter Fuzzing and Interleaving Exploration

To validate and expose concurrency bugs, Themis systematically addresses three necessary conditions: reaching the racy statements, triggering the specific buggy interleaving, and satisfying the failure conditions to manifest symptoms, and achieve the goals in three stages.

First, for the test cases generated with incorrect input parameters, Themis applies directed fuzzing [65] to find the parameter values that can invoke the identified RPCs and reach the racy access statements by setting these statements as fuzzing targets (Stage I). Second, once the access sites are

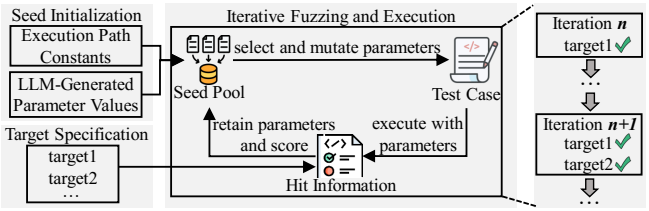


Figure 4: Directed fuzzing process.

reached, Themis manipulates the relevant interleavings of conflicting accesses to expose bugs and checks if the symptoms manifest immediately (Stage II). Finally, because a triggered race may still remain hidden if failure conditions are unsatisfied, Themis launches a follow-up fuzzing phase that inherits the seed pool from Stage I and sets the symptom statements as additional fuzzing targets to drive execution toward observable failures under specific interleavings (Stage III).

5.1 Directed Fuzzing in Stages I and III

Directed fuzzing steers execution toward a prespecified set of target statements, in contrast to coverage-guided fuzzing that aims to maximize coverage. As shown in Figure 4, Themis applies directed fuzzing within an iterative fuzzing loop.

Themis first initializes a seed pool. In each iteration, it selects and mutates a batch of parameter sets (five in our evaluation) and executes the test case with each parameter set. In the first iteration, the LLM-generated test case with its original parameter values is executed directly. To guide this search, Themis relies on a distance metric that quantifies how far the current execution is from the prespecified targets. After every execution, Themis records whether the targets are reached and computes the distance to the target statements. Each parameter set is then scored using both the distance metric and the power scheduling strategy (APS) [66]. Early in the process, APS favors unexplored parameter values, but over time shifts priority toward those closer to the targets. Finally, parameter sets that shorten the distance or cover new code branches are returned to the seed pool, and the highest-scoring ones are selected for the next iteration. Fuzzing continues iteratively until either a parameter set reaches all targets or the time budget is exhausted.

Stages I and III follow the same fuzzing procedure, except that Stage III adds symptom statements as additional fuzzing targets, inherits the final seed pool from Stage I, and executes the test case under all relevant interleavings to ensure the symptom appears only under the buggy interleaving, whereas Stage I executes only under a single interleaving.

To collect runtime data for monitoring test execution and computing distances to the targets, Themis instruments all basic blocks of functions along the target execution path, as well as those of their callees up to seven hops away. If the test executes beyond this range, the distance to the target is considered infinite. This approach efficiently avoids instrumenting the entire system, which could otherwise prevent system startup or significantly degrade performance.

5.2 Stage I: Race-Targeted Parameter Fuzzing

Themis applies directed fuzzing to refine the parameters to reach access sites. When Stage I identifies a parameter set that reaches the accesses, it passes the parameters to Stage II, records the seed pool state, and terminates fuzzing. If the fuzzing time budget is exhausted without reaching the targets, Themis discards the violation tuple. False positives, such as non-communicable RPC pairs or unsatisfiable path conditions between the public interfaces and access sites, are pruned in this manner.

For each test case, Themis sets the fuzzing targets to the RPC call sites and the entry points of the server functions, as well as the entry points of functions $\mathcal{F}$ and the access statements $\mathcal{S}_v$ in the violation tuple $\langle \mathcal{F}, \mathcal{S}_v, T \rangle$. These fuzzing targets ensure that execution follows the intended call chain when they are reached.

The quality of the initial seed pool is critical to the efficiency of fuzzing. In addition to randomly generated seeds used by conventional directed fuzzing, Themis augments the seed pool with LLM-generated parameter value sets and constants extracted from the execution paths. Constants are particularly important, as many path conditions depend on checking a variable's value against a constant, which determines the execution branch and enables triggering the target path.

5.3 Stage II: Interleaving Exploration

Given a parameter set from Stage I that reaches the target accesses under a single interleaving, Themis explores all relevant interleavings for the violation using a signal-and-wait mechanism. For order violations, Themis explores the two possible orders of the two racy accesses. For atomicity violations, Themis explores three interleavings in which one access occurs before, after, or between two consecutive accesses. If the buggy interleaving cannot be reached, the accesses are assumed to be protected by synchronization mechanisms and the violation tuple is pruned. Once all relevant interleavings are exercised, Themis checks the resulting behaviors to determine whether the bug is exposed. If all interleavings are reached but the symptom does not manifest, the parameters and the interleavings are passed to Stage III.

We employ DCatch's [59] signal-and-wait mechanism to control the execution order of accessing statements. A controller coordinates execution by suspending each access immediately before and after it until permission is granted, enabling deterministic exploration of all relevant interleavings.

If the buggy interleaving cannot be enforced by the signal-and-wait mechanism, the violation tuple is pruned. For an order violation, this indicates that only one execution order is feasible; for an atomicity violation, it indicates that the two

consecutive accesses cannot be interleaved by another access. In both cases, the accesses are likely guarded by synchronization mechanisms missed by static analysis.

Themis executes the test case under all relevant interleavings and checks their behaviors. For an order violation between two accesses, Themis verifies that the identified symptom manifests only under one specific order, while the opposite order produces no symptom. For an atomicity violation involving three accesses, Themis checks that the symptom appears only when one access occurs between the other two consecutive accesses, whereas executions with that access occurring before or after the pair produce no error. For thread-unsafe objects (TO), Themis does not directly trigger bugs in standard Java thread-unsafe classes, but it verifies that the execution path is reachable and interleavings are adjustable; it returns the potential violations to users for final determination. In our evaluation, all the detected TO violations are confirmed as true positives with observable symptoms.

5.4 Stage III: Symptom-Targeted Parameter Fuzzing

Themis reuses the seed pool and the parameters from Stage I to initiate a new fuzzing process, adjusting the parameters to satisfy failure conditions and expose the associated symptoms. To determine whether the symptoms are reached, Themis includes the symptom statements as additional fuzzing targets, in addition to all Stage-I targets. Stage III is allocated a separate time budget, and the APS strategy accounts only for the elapsed time within this stage. For each parameter value set generated during fuzzing, Themis first checks whether the RPC and racy access statements are reached under a particular interleaving, as in Stage I. If so, Themis explores all relevant access interleavings produced by the signal-and-wait mechanism and checks behaviors as in Stage II. Fuzzing terminates once all relevant interleavings are exercised and only the buggy interleaving triggers the symptom, which is consistent with the behavior-checking criterion in Stage II; otherwise, it continues until the time budget is exhausted. When the budget is exhausted, Themis prunes the violation. Benign races that reach the accesses but never manifest symptoms are eliminated in this manner.

Themis initializes Stage III by inheriting the final seed pool from Stage I and using its final parameter set as the starting point. Since Stage I already ensures that the accesses are reached, only the failure conditions remain unsatisfied. As the seed pool accumulates increasingly effective parameters during Stage I, Stage III begins with parameter values that are near the remaining targets, enabling efficient exploration of error-triggering path conditions.

Themis retains all fuzzing targets from Stage I and adds the symptom statements as additional targets. For implicit errors, reaching potential symptom statements does not necessarily trigger the error. For example, in Figure 1a, accessing line 537

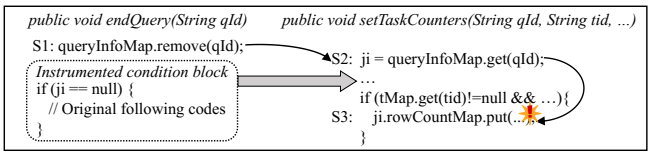


Figure 5: Hive-29150 bug example.

does not by itself trigger the error; the error occurs only when `task` is `null`. To address this, Themis instruments checks that transform error-triggering conditions into explicit control-flow branches and treats them as fuzzing targets. For null pointer dereferences in NP, UR, and MR patterns, it inserts pre-dereference null checks (e.g., `if (var == null)`). For out-of-bounds errors on array-like variables (e.g., `ArrayList`) in OB pattern, it adds index range checks (e.g., `if (i < 0 || i >= arr.length)`), thereby making these errors explicit.

Case Study. In Figure 5, removing an entry from `queryInfoMap` at S1 before it is retrieved at S2 causes `ji` to be `null`. Themis instruments an `if`-condition block after S2 to guide the fuzzer in finding a parameter value that causes `ji` to be null. With Stage I setting `tid` to an invalid key, the error at S3 is not triggered. Stage III refines `tid` to a valid key, exposing the bug.

6 Evaluation

We have implemented a prototype of Themis consisting of approximately 6.5K lines of Java code for static analysis and fuzzing, and 700 lines of Python code for building the LLM-based test generation. The static analysis and fuzzing instrumentation are built on top of the Soot framework [20], and RPC extraction is implemented using RPCBridge [30]. We employ Claude-4 [13] as the backbone model for all LLM-based tasks in Themis, due to its demonstrated strengths in code comprehension and program synthesis. We build the directed fuzzing engine by integrating the techniques of SelectFuzz [65] into the Java fuzzing framework JQF [70]. The fuzzing time budget is set to 30 minutes.

Target Systems. We evaluate Themis on the latest releases of eight popular cloud systems, including Alluxio [6], Hadoop Common (Hadoop) [15], HBase [11], HDFS [16], Hive [17], MapReduce [18], Tez [21], and YARN [23]. More details of these systems are in Appendix A.5. These systems are representative of open-source distributed platforms and incorporate four RPC frameworks, i.e., gRPC [14], Hadoop IPC [15], Netty [19], and Thrift [22].

Measurement of New Bug Discovery. The primary measure of Themis as a detection tool is its ability to uncover new bugs. We evaluate Themis on the latest releases of the eight systems, using the *number of violations* and the *number of bugs* to quantify its detection capability. The *number of violations* counts unique violations, while the *number of bugs* counts groups of violations that could be fixed by the same

patch, including atomicity violations resolved by adding a lock to serialize the same accesses and order violations addressed by enforcing specific execution orders.

New Bug Confirmation. We have reported all the newly discovered bugs to the developers. However, bug confirmation through Apache Jira [12], which is the bug tracking system used by the systems we evaluated, typically takes a long time. For bugs not confirmed by developers, we validate them as true bugs in two ways: (1) Themis demonstrates that the bug can be triggered via public interfaces under feasible interleavings and produces observable symptoms; and (2) we manually inspect source-code comments, documentation, and system manuals to confirm that users are neither instructed nor expected to enforce a specific operation order or maintain atomicity for the racy accesses.

Known Bug Detection. We review reported bugs from prior studies [49, 59–61]. We exclude state-machine–related bugs, as they are typically detectable through other approaches [36, 46, 47]. After filtering, we collect six known bugs in the eight targeted systems. We run static analysis on the buggy system versions to determine whether these bugs are covered. For the covered ones, we further evaluate whether Themis can detect them by applying LLM-based test generation on the extracted paths, followed by directed fuzzing and interleaving exploration to trigger the bugs.

6.1 Detection Results

Table 3 and Table 4 present the detection results for Themis. Appendix A.5 shows the descriptions of all detected bugs. Specifically, Themis detects **204** unique violations (**198** of which are new), corresponding to **58** bugs (**52** of which are new). We have reported all the 52 new bugs to developers, and six have been confirmed as of the paper's publication. Such confirmations typically take months, as they rely on maintainers' manual review and deliberation.

In contrast to prior dynamic detection approaches such as DCatch [59] and CloudRaid [60, 61], which uncover only a small number of bugs, most of which are already known, Themis's static-driven approach reveals dozens of previously unknown ones. Because DCatch and CloudRaid are not open source, we cannot apply them to the latest versions of the evaluated systems for direct comparison. Experiments on the known-bug benchmark show that all the six known bugs are successfully detected with no false negatives.

170 out of 204 detected violations (83.3%) exhibit implicit symptoms. Among them, 141 cases belong to the UR pattern, which occurs when functions access variables before they are initialized. These 141 violations are densely merged when reporting bugs. For example, the Yarn-11846 bug consolidates 36 unique UR violations because they involve the same initialization function and shared variable, with each violation occurring between the initialization function and another function reading the shared variable.

Table 3: Number of detected violations. Superscript indicates the number of violations that correspond to known bugs.

System	EO	EA	NP	OB	UR	MR	TO	Total
Alluxio	0	0	0	0	2	0	0	**2**
Hadoop	1	0	1	0	0	0	0	**2**
HBase	5^{3}	3^{1}	1	0	36	1	2	**48**^{4}
HDFS	3	0	0	0	1	3	1	**8**
Hive	2	0	0	0	10	1	0	**13**
MapReduce	7^{2}	3	3	2	19	3	2	**39**^{2}
Tez	1	0	0	0	2	0	0	**3**
Yarn	9	0	4	0	71	2	3	**89**
Total	**28**^{5}	**6**^{1}	**9**	**2**	**141**	**10**	**8**	**204**^{6}

Table 4: 52 new bugs and 6 known bugs detected by Themis. Six confirmed newly discovered bugs are **bolded**. '#Vio' refers to the number of violations merged into this bug.

#	Bug ID	Pattern	#Vio	#	Bug ID	Pattern	#Vio
1	**Alluxio-18738**	**UR**	**2**	31	MapReduce-7499	UR	7
2	Hadoop-19462	EO	1	32	MapReduce-7510	UR	10
3	**Hadoop-19458**	**NP**	**1**	33	MapReduce-7515	UR	2
4	HBase-29525	EO	2	**34**	**MapReduce-7507**	**MR**	**2**
5	HBase-29516	EA	2	35	MapReduce-7518	MR	1
6	HBase-29607	NP	1	36	MapReduce-7512	TO	1
7	**HBase-29529**	**UR**	**17**	37	MapReduce-7513	TO	1
8	HBase-29530	UR	5	38	Tez-4644	EO	1
9	HBase-29538	UR	6	**39**	**Tez-4645**	**UR**	**2**
10	HBase-29539	UR	8	40	Yarn-11769	EO	2
11	HBase-29606	MR	1	41	Yarn-11770	EO	2
12	HBase-29510	TO	1	42	Yarn-11772	EO	3
13	HBase-29595	TO	1	43	Yarn-11773	EO	2
14	HDFS-17726	EO	1	44	Yarn-11771	NP	2
15	HDFS-17727	EO	1	45	Yarn-11847	NP	2
16	HDFS-17879	EO	1	46	Yarn-11846	UR	36
17	HDFS-17823	UR	1	47	Yarn-11849	UR	22
18	HDFS-17734	MR	3	48	Yarn-11852	UR	9
19	HDFS-17822	TO	1	49	Yarn-11853	UR	4
20	Hive-29143	EO	2	50	Yarn-11860	MR	2
21	**Hive-29151**	**UR**	**6**	51	Yarn-11851	TO	1
22	Hive-29156	UR	4	52	Yarn-11859	TO	2
23	Hive-29150	MR	1	53	HBase-4539	EO	1
24	MapReduce-7498	EO	2	54	HBase-5780	EO	1
25	MapReduce-7501	EO	2	55	HBase-6070	EO	1
26	MapReduce-7514	EO	1	56	HBase-4729	EA	1
27	MapReduce-7511	EA	3	57	MapReduce-4099	EO	1
28	MapReduce-7516	NP	2	58	MapReduce-4637	EO	1
29	MapReduce-7529	NP	1	**Total**		**/**	**204**
30	MapReduce-7509	OB	2				

The newly detected bugs exhibit root causes similar to those of previously known bugs. For example, in HBase-4729, concurrent calls to `split()` and `unassign()` on the same HMaster cause an order violation that raises `NodeExistsException`, while in MapReduce-7514, concurrent calls to `purgeLogsOlderThan()` and `getJournaledEdits()` on the same journal server cause an order violation that raises `NewerTxnIdException`. The results highlight a common pitfall: developers often overlook that server functions may be invoked on the same class instance by multiple RPC clients, and this leads to simple but harmful races, where two functions accessing the same shared variable are invoked concurrently by different clients.

The shared variables span multiple types. 45 of the 204 violations (22.1%) involve collections, 2 involve arrays, and 2 involve IO/NIO objects. In particular, the OB, MR, and TO patterns arise only from arrays, collections, and IO/NIO objects. The two OB cases are order violations. The 10 MR cases

include 2 atomicity violations (e.g., MapReduce-7507 in Figure 1a) and 8 order violations (e.g., Hive-29150 in Figure 5). Among the 8 TO cases, half (4/8) manifest as `ConcurrentModificationException`, which occurs when a collection is modified concurrently during iteration. The other 4 TO cases arise from concurrent operations overwriting each other. For example, in Yarn-11859, concurrent `add` and `clear` operations on the same `io.File` object race, with the `clear` overwriting the `add`, causing the newly added content to be lost and resulting in an exception.

Seven out of 34 violations in the EO and EA pattern propagate effects across different nodes via RPC. Such cases occur when a server-side race causes the RPC call to return an unexpected value, which in turn triggers a client-side error.

Among the 204 violations, 23 are atomicity violations (corresponding to 16 bugs) and 181 are order violations (corresponding to 42 bugs). Particularly, all the 141 UR violations fall under order violations, contrasting with local concurrency bugs where atomicity violations dominate.

The average **detection time** of Themis is 34.6 minutes per system for static analysis, 10.2 minutes per violation for LLM-based test generation, and 6.1 minutes per violation for parameter fuzzing with interleaving exploration. The average **token cost** per violation is 487K. More detailed analysis is presented in Appendix A.6.

6.2 Module-Specific Result Analysis

In this section, we evaluate the effectiveness of Themis's three modules and provide a detailed analysis of the results.

6.2.1 Static Analysis

Using violation analysis rules, Themis identifies **685** potential violation tuples (633 server-side and 52 client-side). After applying error-pattern matching, Themis detects **239** unique violations (234 server-side and 5 client-side). Note that, the numbers after error-pattern matching no longer double-count subsumed order violations. Moreover, 4 violation tuples produce 8 violations, each corresponding to two symptoms.

We randomly sample several pruned violations and manually confirm that they are all benign races. For instance, in Figure 1a, the accesses to `jvmMap` at lines 551 and 582 constitute a benign race, because `jvmMap` is of the `ConcurrentHashMap` type, designed to allow safe concurrent operations.

Client-side violations are much fewer than server-side violations for two main reasons. First, only a small fraction of RPC-reachable variables reside on the client, because many RPC calls are used solely to invoke server services without passing parameters or returning values. Second, client-side violations involve RPC accesses that behave like local procedure calls (LPCs); developers, familiar with local races, often add synchronization and error handling, which our analyses in Sections 3.2 and 3.3 handle to prevent false positives.

6.2.2 Test Generation

Themis successfully generates error-free test cases for **228** out of 239 violations. After best-effort manual checking on the remaining 11, we confirmed six as false negatives by manually constructing valid test cases. The other five remain inconclusive, as we could not construct a test case but cannot definitively prove infeasibility due to the large search space. Among the 228 generated test cases, 218 are constructed from the nearest public interfaces (i.e., the first selected public-interface pair already succeeds), while 10 are constructed from top-level public interfaces. For these 10 cases, the number of tried public-interface pairs ranges from 2 to 12. Even with appropriate public interfaces, test generation often requires multiple generation–repair cycles. 200 of the 228 test cases are produced by generation without test suites, while the remaining 28 leverage existing test suites for call sequence construction. More details are presented in Appendix A.7.

We conduct a qualitative analysis of the call sequences produced by Themis and find that they exhibit substantial structural diversity that cannot be captured by a small set of canonical workflows. Even for RPC initialization, we observe both explicit initialization patterns (e.g., direct server setup) and implicit activation through high-level workflows (e.g., job submission or table creation), often coexisting within the same system. Beyond RPC initialization, call-sequence construction requires heterogeneous state-construction strategies, including explicit API-based configuration, waiting for system-level state convergence, and manipulation of shared collections. This diversity explains our design choice of using an LLM rather than template-based test synthesis.

We study the six confirmed false negatives and find two dominant causes. First, semantic hallucinations introduce fabricated methods or mismatched types in the initial harness, often caused by the model relying on prior knowledge from historical or related systems. Such hallucinations derail the generation–repair loop, as repeated repair attempts focus on correcting invalid APIs and exhaust the repair budget without converging to a feasible test. Second, some violations depend on call sequences whose feasibility hinges on external services or third-party systems (e.g., HBase–ZooKeeper interactions) that are not visible to the LLM. Since Themis targets a single system and cannot practically incorporate all external dependencies into the prompt context, these cross-system analyses are left for future work.

6.2.3 Parameter Fuzzing and Interleaving Exploration

As shown in Table 4, Themis exposes **204** true violations. The detailed results of the three stages are discussed below.

80 of the 228 generated test cases require refinement before the racy access sites can be exercised. Stage I prunes 17 violations (12 true negatives and 5 false negatives). Among the 12 true negatives, 8 cases belong to uncommunicable RPC pairs. 3 cases are caused by conflicting conditions that

require different values of the same variable to trigger the racy accesses. For instance, one pruned order violation requires variable `nn` to be in `OBSERVER_STATE` to trigger one access and in `STANDBY_STATE` to trigger the other, rendering the violation impossible. Another case fails to reach the accesses due to an incomplete call sequence; however, it is a true negative because a custom lock enforces the atomicity of the accesses. Five false negatives are caused by incorrect call sequences. We manually construct valid test cases for these five cases to confirm them as false negatives.

Stage II exposes 167 violations and prunes 2 false positives. For all the 167 exposed violations, the target symptom is triggered directly after checking behaviors under all relevant interleavings. Two violations are pruned due to synchronization (one atomicity and one order violation). In the atomicity case, consecutive accesses are protected by a custom lock not recognized by the static analysis. In the order case, one access busy-waits on a flag set by the other, enforcing a fixed execution order that static analysis cannot identify either.

42 violations proceed to Stage III due to symptoms not manifesting immediately under the buggy interleaving. Stage III exposes 37 true bugs requiring additional fuzzing for triggering errors (e.g., Hive-29150, Figure 5), while 5 benign races are pruned for failing to satisfy failure conditions, despite matching error patterns in static analysis (e.g., lines 521–551–533, Figure 1a).

6.3 LLM-Only and Prompt Variants Study

To validate the necessity and effectiveness of our "LLM + guidance + checking" design, we conduct three comparative experiments representing alternative LLM-based designs. We summarize the results below, and we report detailed settings and results in Appendix A.8.

LLM-only Bug Detection without Static Analysis Guidance. To evaluate the effectiveness of LLM-only detection, we ask the LLM to identify distributed concurrency bugs directly from the entire codebase. We consider two configurations, including the zero-shot and the few-shot prompt. The results show that the LLM fails to identify any real distributed concurrency bug, indicating that LLMs alone are insufficient for discovering such bugs in large systems, as they cannot systematically identify RPC communication or distinguish harmful bugs from benign races.

Prompt Variation of LLM-based Test Generation. We study how prompt design affects LLM-based test generation by varying the instruction and context components of the prompt. Due to the high cost of generation, we evaluate 10 randomly selected violations. Without structured, multi-step instructions, the LLM succeeds in only 1/10 cases, frequently missing essential steps such as RPC initialization and system-specific call sequence construction. Providing richer context improves recall (9/10 with the full codebase and 7/10 with test suites) but incurs a token cost approximately four times that of Themis and introduces distracting or misleading information. Overall, these results show that both instruction structure and context curation are critical: Themis's default prompt, which combines guided reasoning with focused context, achieves the best balance between accuracy and efficiency.

LLM-only Parameter Tuning with Runtime Feedback. We evaluate whether an LLM can replace the fuzzer in the violation-triggering phase. It iteratively refines input parameters based on execution feedback collected from our instrumentation module used for fuzzing. The result shows that LLM-only parameter tuning achieves a significantly lower success rate (i.e., 20%) and, even when successful, incurs substantially higher token costs (i.e., 324K tokens on average), making it an impractical replacement for directed fuzzing.

6.4 Discussion of False Negatives

No tool is perfect. Although we carefully design each module to maximize real-bug coverage and Themis provides strong benefits in detecting tens of previously unknown bugs, Themis may still incur false negatives, as do other bug detection tools. Static analysis may miss bugs because we target observable symptoms such as exceptions, error logs, and system aborts, while excluding hangs and silent failures. LLM-based test generation may also introduce false negatives due to hallucinations that yield infeasible call sequences, as discussed in Section 6.2.2. Fuzzing, being a heuristic search technique, can likewise miss bugs, although we did not observe such cases under our current settings and optimizations (e.g., increased time budgets and an improved seed pool).

Beyond theoretical considerations, empirically measuring false negatives is challenging. A common practice is to evaluate whether a tool misses bugs from a known benchmark set. Following this approach, we make our best effort to include known bugs from prior work in our evaluation. As reported in Section 6.1, Themis successfully detects all of these known bugs without omission.

7 Conclusion

This paper presents Themis, a new framework for detecting distributed concurrency bugs. By integrating static RPC-driven race detection, LLM-based test generation, and parameter fuzzing and interleaving exploration, Themis effectively exposes real-world bugs. Evaluated on eight widely used distributed systems, Themis uncovers 58 bugs, including 52 previously unknown ones.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Matthew Caesar, for their insightful comments and feedback. This work was supported by ShanghaiTech.

References

[1] Java string analyzer. https://www.brics.dk/JSA/, 2003.

[2] AFL. https://github.com/google/AFL, 2021.

[3] Honggfuzz. https://github.com/google/honggfuzz, 2024.

[4] libFuzzer. https://llvm.org/docs/LibFuzzer.html, 2024.

[5] Syzkaller. https://github.com/google/syzkaller, 2024.

[6] Alluxio. https://documentation.alluxio.io/os-en, 2025.

[7] Amazon DynamoDB outage CNN report. https://edition.cnn.com/2025/10/25/tech/aws-outage-cause, 2025.

[8] Amazon DynamoDB outage impact. https://www.yahoo.com/news/world/article/aws-outage-list-of-popular-apps-and-websites-affected-by-global-service-disruption-200714556.html, 2025.

[9] Amazon DynamoDB outage loss. https://www.cybcube.com/news/insurance-loss-estimate-for-aws-amazonk-outage, 2025.

[10] Amazon DynamoDB outage report. https://aws.amazon.com/message/101925/, 2025.

[11] Apache HBase. http://hbase.apache.org, 2025.

[12] Apache Jira. https://issues.apache.org/jira/secure/Dashboard.jspa, 2025.

[13] Claude. https://claude.ai/, 2025.

[14] gRPC. https://grpc.io/, 2025.

[15] Hadoop Common. https://hadoop.apache.org/, 2025.

[16] HDFS. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html, 2025.

[17] Hive. https://hive.apache.org/, 2025.

[18] MapReduce. https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html, 2025.

[19] Netty. https://netty.io/, 2025.

[20] Soot. http://soot-oss.github.io/soot/, 2025.

[21] Tez. https://tez.apache.org/, 2025.

[22] Thrift. https://thrift.apache.org/, 2025.

[23] Yarn. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html, 2025.

[24] Sam Blackshear, Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. Racerd: compositional static race detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–28, 2018.

[25] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*, pages 2329–2344, 2017.

[26] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NeurIPS'20)*, pages 1877–1901, 2020.

[27] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. Muzz: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *29th USENIX Security Symposium (SEC'20)*, pages 2325–2342, 2020.

[28] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*, pages 2095–2108, 2018.

[29] Aske Simon Christensen, Anders Møller, and Michael I Schwartzbach. Precise analysis of string expressions. In *International Static Analysis Symposium (SAS'03)*, pages 1–18. Springer, 2003.

[30] Baoquan Cui, Rong Qu, Zhen Tang, and Jian Zhang. Static analysis of remote procedure call in java programs. In *2025 IEEE/ACM 45th International Conference on Software Engineering (ICSE'25)*, pages 1–12. IEEE, 2025.

[31] Dawson Engler and Ken Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP'03)*, page 237–252, 2003.

[32] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE'23)*, pages 31–53. IEEE, 2023.

[33] 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)*, 2020.

[34] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*, pages 213–223, 2005.

[35] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*, pages 66–83, 2021.

[36] Roland Groz, Adenilso Simao, Alexandre Petrenko, and Catherine Oriat. Inferring finite state machines without reset using state identification sequences. In *IFIP International Conference on Testing Software and Systems (ICTSS'15)*, pages 161–177. Springer, 2015.

[37] Fangming Gu, Qingli Guo, Lian Li, Zhiniang Peng, Wei Lin, Xiaobo Yang, and Xiaorui Gong. Comrace: detecting data race vulnerabilities in com objects. In *31st USENIX Security Symposium (SEC'22)*, pages 3019–3036, 2022.

[38] Dae R Jeong, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. Segfuzz: Segmentizing thread interleaving to discover kernel concurrency bugs through fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP'23)*, pages 2104–2121. IEEE, 2023.

[39] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Context-sensitive and directional concurrency fuzzing for data-race detection. In *Network and Distributed Systems Security Symposium (NDSS'22)*, 2022.

[40] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11)*, pages 389–400, 2011.

[41] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. Automated concurrency-bug fixing. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, pages 221–236, 2012.

[42] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'09)*, pages 13–22, 2009.

[43] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. Fast and accurate static data-race detection for concurrent programs. In *International Conference on Computer Aided Verification (CAV'07)*, pages 226–239, 2007.

[44] Baris Kasikci, Cristian Zamfir, and George Candea. Racemob: Crowdsourced data race detection. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*, pages 406–422, 2013.

[45] Zhifeng Lai, Shing-Chi Cheung, and Wing Kwong Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, pages 235–244, 2010.

[46] David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 2002.

[47] David Lee and Mihalis Yannakakis. Testing finite-state machines: State identification and verification. *IEEE Transactions on Computers*, 43(3):306–320, 2002.

[48] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. Constraint-guided directed greybox fuzzing. In *30th USENIX Security Symposium (SEC'21)*, pages 3559–3576, 2021.

[49] Tanakorn Leesatapornwongsa, Jeffrey F Lukman, Shan Lu, and Haryadi S Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, pages 517–530, 2016.

[50] Ondřej Lhoták and Laurie Hendren. Scaling java points-to analysis using spark. In *International Conference on Compiler Construction (CC'03)*, pages 153–169. Springer, 2003.

[51] Ao Li, Shan Lu, Suman Nath, Rohan Padhye, and Vyas Sekar. Exchain: Exception dependency analysis for root cause diagnosis. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI'24)*, pages 2047–2062, 2024.

[52] Guangpu Li, Haopeng Liu, Xianglan Chen, Haryadi S. Gunawi, and Shan Lu. Dfix: automatically fixing timing bugs in distributed systems. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*. ACM, 2019.

[53] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*, pages 162–180, 2019.

[54] Yuwei Li, Shouling Ji, Chenyang Lyu, Yuan Chen, Jianhai Chen, Qinchen Gu, Chunming Wu, and Raheem Beyah. V-fuzz: Vulnerability prediction-assisted evolutionary fuzzing for binary programs. *IEEE Transactions on Cybernetics*, 52(5):3745–3756, 2020.

[55] Hongliang Liang, Lin Jiang, Lu Ai, and Jinyi Wei. Sequence directed hybrid fuzzing. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER'20)*, pages 127–137. IEEE, 2020.

[56] Bozhen Liu, Peiming Liu, Yanze Li, Chia-Che Tsai, Dilma Da Silva, and Jeff Huang. When threads meet events: efficient and precise static race detection with origins. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'21)*, pages 725–739, 2021.

[57] Changming Liu, Deqing Zou, Peng Luo, Bin B Zhu, and Hai Jin. A heuristic framework to detect concurrency vulnerabilities. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC'18)*, pages 529–541, 2018.

[58] Haopeng Liu, Yuxi Chen, and Shan Lu. Understanding and generating high quality patches for concurrency bugs. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*, pages 715–726, 2016.

[59] Haopeng Liu, Guangpu Li, Jeffrey F. Lukman, Jiaxin Li, Shan Lu, Haryadi S. Gunawi, and Chen Tian. Dcatch: Automatically detecting distributed concurrency bugs in cloud systems. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, page 677–691, 2017.

[60] Jie Lu, Feng Li, Lian Li, and Xiaobing Feng. Cloudraid: hunting concurrency bugs in the cloud via log-mining. In *Proceedings of the 2018 26th ACM Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'18)*, pages 3–14, 2018.

[61] Jie Lu, Feng Li, Chen Liu, Lian Li, Xiaobing Feng, and Jingling Xue. Cloudraid: Detecting distributed concurrency bugs via log mining and enhancement. *IEEE Transactions on Software Engineering*, 48(2):662–677, 2020.

[62] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A Popa, and Yuanyuan Zhou. Muvi: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*, pages 103–116, 2007.

[63] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*, pages 329–339, 2008.

[64] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, page 37–48, 2006.

[65] Changhua Luo, Wei Meng, and Penghui Li. Selectfuzz: Efficient directed fuzzing with selective path exploration. In *2023 IEEE Symposium on Security and Privacy (SP'23)*, pages 2693–2707. IEEE, 2023.

[66] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S Foster, and Michael Hicks. Directed symbolic execution. In *Proceedings of the 18th International Conference on Static Analysis (SAS'11)*, pages 95–111. Springer, 2011.

[67] Suvam Mukherjee, Pantazis Deligiannis, Arpita Biswas, and Akash Lal. Learning-based controlled concurrency testing. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–31, 2020.

[68] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*, pages 308–319, 2006.

[69] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. Binary-level directed fuzzing for use-after-free vulnerabilities. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID'20)*, pages 47–62, 2020.

[70] Rohan Padhye, Caroline Lemieux, and Koushik Sen. Jqf: Coverage-guided property-based testing in java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'19)*, pages 398–401, 2019.

[71] Michael Pradel and Thomas R Gross. Fully automatic and precise detection of thread safety violations. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*, pages 521–530, 2012.

[72] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*, page 320–331, 2006.

[73] Malavika Samak and Murali Krishna Ramanathan. Synthesizing tests for detecting atomicity violations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*, pages 131–142, 2015.

[74] Malavika Samak, Murali Krishna Ramanathan, and Suresh Jagannathan. Synthesizing racy tests. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, page 175–185, 2015.

[75] Bogdan Alexandru Stoica, Shan Lu, Madanlal Musuvathi, and Suman Nath. Waffle: exposing memory ordering bugs efficiently with active delay injection. In *Proceedings of the 18th European Conference on Computer Systems (EuroSys'23)*, pages 111–126, 2023.

[76] Xin Tan, Yuan Zhang, Jiadong Lu, Xin Xiong, Zhuang Liu, and Min Yang. Syzdirect: Directed greybox fuzzing for linux kernel. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS'23)*, pages 1630–1644, 2023.

[77] Valerio Terragni and Shing-Chi Cheung. Coverage-driven test code generation for concurrent classes. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*, pages 1121–1132, 2016.

[78] Valerio Terragni, Mauro Pezzè, and Francesco Adalberto Bianchi. Coverage-driven test generation for thread-safe classes via parallel and conflict dependencies. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST'19)*, pages 264–275. IEEE, 2019.

[79] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: static race detection on millions of lines of code. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'07)*, pages 205–214, 2007.

[80] Chao Wang, Mahmoud Said, and Aarti Gupta. Coverage guided systematic concurrency testing. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 221–230, 2011.

[81] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *Proceedings of the 36th International Conference on Neural Information Processing Systems (NeurIPS'22)*, 35:24824–24837, 2022.

[82] Dylan Wolff, Zheng Shi, Gregory J Duck, Umang Mathur, and Abhik Roychoudhury. Greybox fuzzing for concurrency testing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'24)*, pages 482–498, 2024.

[83] Ming Yuan, Bodong Zhao, Penghui Li, Jiashuo Liang, Xinhui Han, Xiapu Luo, and Chao Zhang. Ddrace: Finding concurrency uaf vulnerabilities in linux drivers with directed fuzzing. In *32th USENIX Security Symposium (SEC'23)*, pages 2849–2866, 2023.

[84] Minjian Zhang, Daniel Wee Soong Lim, Mosaad Al Thokair, Umang Mathur, and Mahesh Viswanathan. Efficient timestamping for sampling-based race detection. *Proceedings of the ACM on Programming Languages*, 9(PLDI):150–175, 2025.

[85] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. Conseq: detecting concurrency bugs through sequential errors. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*, page 251–264, 2011.

[86] Wei Zhang, Chong Sun, and Shan Lu. Conmem: detecting severe concurrency bugs through an effect-oriented

approach. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*, page 179–192, 2010.

[87] Huan Zhao, Dylan Wolff, Umang Mathur, and Abhik Roychoudhury. Selectively uniform concurrency testing. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'25)*, pages 1003–1019, 2025.

Appendix A

A.1 Access Sites and Read and Write Classification

Table 5 shows the access sites and the corresponding read and write types. The read and write classification is extended to collection and IO/NIO objects via manual labeling of key methods in standard Java packages (e.g., `io`, `nio`, `util.zip`). For instance, `ArrayList.add()` is marked as a write, whereas `ConcurrentHashMap.remove()` is marked as both read and write due to its retrieval-and-deletion semantics.

A.2 Completeness of the Rules

Rules ❶–❺ collectively cover all possible concurrency scenarios, which fall into either order or atomicity violations. Order violations arise when two distinct accesses to the same variable $\{s_1, s_2\}$ interleave without an expected consecutive order. Such interleaving can occur in two ways: 1) by concurrently invoking two different functions that respectively reach s_1 and s_2 (captured by Rule ❶); or 2) by a single function that reaches two accesses and materializes concurrency via asynchronous mechanisms (captured by Rule ❹). In contrast, atomicity violations arise when two consecutive accesses $\{s_1, s_2\}$ are overlapped by an additional access s_3, forming one of the four buggy interleaving patterns (RWR, WWR, WRW, RWW). This overlap may be introduced 1) across two different functions (Rule ❷); 2) by repeated concurrent invocations of a single function that self-interleave (Rule ❸); or 3) by asynchronous execution within a single function that overlaps consecutive accesses with a concurrent access (Rule ❺). Note that repeated invocations of the same function (Rule ❸) cannot produce order violations, since there is no predefined ordering between a function and itself.

A.3 Details of Error Pattern Matching

Table 6 summarizes the characteristics of the seven error patterns, including their violation types, shared object types, interleaving patterns, and corresponding symptoms. While

Table 5: Access sites that access shared objects. write_var denotes a written variable; `read_var` denotes a read variable; base_var/para_var denotes a variable that needs further analysis in the callee; *cast_op is type casting; *bi_op is a binary operation; *un_op is a unary operation; *cmp_op is a comparison operation; Class is a static class.

Access sites	
`write_var` = `read_var`;	`base_var`.func(...);
`write_var` = *un_op `read_var`;	Class.func(`para_var`, ...);
`write_var` = `read_var1` *bi_op `read_var2`;	`base_var`.func(`para_var`, ...);
`write_var` = *cast_op `read_var`;	if(`read_var`) *cmp_op `read_var2`);
`write_var` = `base_var`.func(...);	switch(`read_var`);
`write_var` = Class.func(`para_var`, ...);	`hashMap`.remove(`read_var`);
`write_var` = `base_var`.func(`para_var`, ...);	`bufferedReader`.readLine()

explicit symptoms, namely explicit order and atomicity violations, are described in Section 3.3, we detail the implicit symptoms in the following.

Null Pointer Dereference (NP). This pattern occurs when dereferencing a shared object whose value is `null`. Both order and atomicity violations can lead to null pointer dereferences. For order violations, the pattern considers tuples with one write and one read, where the write assigns the shared variable to `null` (i.e., a non-instantiation write operation), and the read dereferences the variable via a method call or field access. For example, in the Yarn-11847 bug, if an RPC client passes `null` to an RPC server function, the shared variable `rmAdminProxy` is assigned `null`. A subsequent call to `getGroupsForUser()` then dereferences `rmAdminProxy`, resulting in `NullPointerException`. For atomicity violations, the pattern considers tuples in which (1) a non-instantiation write is interleaved with two consecutive accesses of the read–read or write–read type (i.e., RWR or WWR), causing the final read to operate on `null` assigned by the intermediate write and throw an exception; or (2) a read is interleaved with two consecutive writes (i.e., WRW), where the first write is non-instantiation and nullifies the shared variable. For example, in the Yarn-11771 bug, although the RPC server function first checks whether the instance variable `activeServices` is `null` and then dereferences it, another concurrently executing RPC server function can still set `activeServices` to `null` between the check and the dereference, exhibiting an RWR interleaving and leading to `NullPointerException`.

Out-of-Bounds Access (OB). This pattern arises when a shared array-like object is concurrently accessed, causing one of the accesses to fall outside its valid bounds. Array-like objects include arrays provided by the `java.util.Arrays` package and Java library classes that support array-like access (e.g., `java.util.ArrayList`). Both order and atomicity violations can lead to out-of-bounds exceptions. For order violations, the OB symptom may arise at either write in write–write (W+W) violations, but only at the read in write–read (W+R) violations. For atomicity violations, an intermediate write from another concurrent execution may modify the shared

Table 6: Characteristics of the seven error patterns. 'W+R' and 'W+W' denote write–read and write–write order violations, respectively, regardless of the access order. '$\widehat{\ }$' marks the access that triggers the symptom. For patterns where the symptom occurs after further propagation, no access is marked. 'Inst.' and 'non-inst.' distinguish instantiation and non-instantiation operations. 'Retr.' denotes a retrieval operation for array and collection objects.

Pattern	Violation Type	Shared Object Type	Interleaving Pattern	Symptom Statement
EO	O	All	W+W/W+R	Explicit symptom
EA	A		RWR/WWR/WRW/RWW	
NP	O/A		W(non-inst.)$\widehat{\text{R}}$/ RW(non-inst.)$\widehat{\text{R}}$/ WW(non-inst.)$\widehat{\text{R}}$/ W(non-inst.)$\widehat{\text{RW}}$	Dereference
OB	O/A	Array-like	$\widehat{\text{W}}$+W/W+$\widehat{\text{R}}$/ RW$\widehat{\text{R}}$/WW$\widehat{\text{R}}$/RW$\widehat{\text{W}}$/W$\widehat{\text{RW}}$	
UR	O	All	$\widehat{\text{RW}}$(inst.)	
MR	O/A	Collection and array	W+R(retr.)/ RWR(retr.)/ WWR(retr.)/ WR(retr.)W	
TO	A	Thread unsafe collections and IO/NIO	$\widehat{\text{W}}$+$\widehat{\text{W}}$/$\widehat{\text{W}}$+$\widehat{\text{R}}$	/

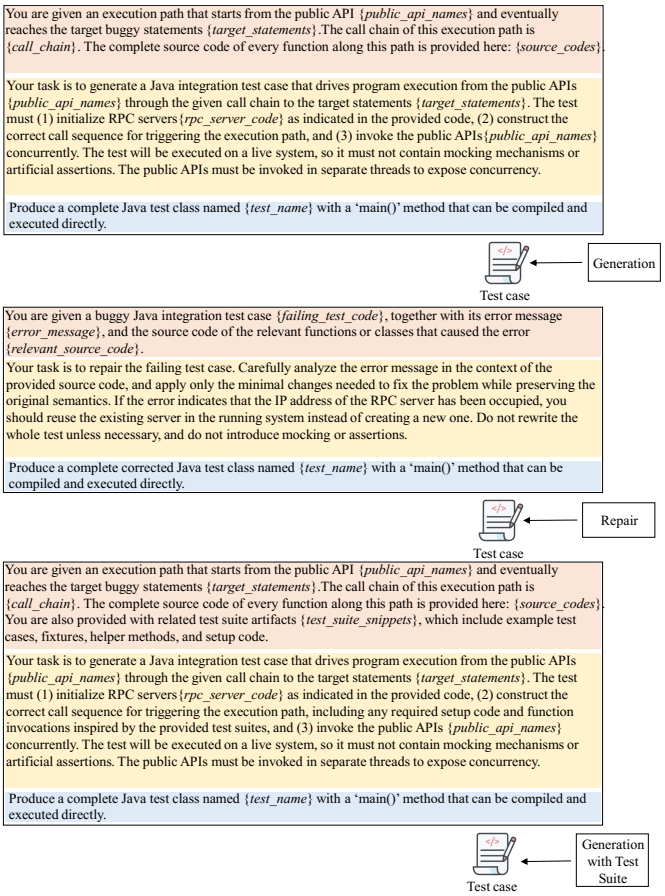


Figure 6: The prompt designed for three test generation prompts. Knowledge refers to the knowledge provided to the LLM. Task describes the task Themis asks the LLM to do. Format describes the output (test case) format.

array-like object between two consecutive accesses, namely RWR, WWR, or RWW. Consequently, regardless of whether the final access is a read or a write, an out-of-bounds index introduced by the intermediate write can still raise the exception. Another scenario arises in the WRW interleaving pattern, where a read is interleaved between two writes and accesses an out-of-bounds index only during this intermediate access.

Uninitialized Read (UR). We observe that a portion of RPC servers includes initialization functions that set the initial values of variables. However, these functions may execute after other RPC server functions, causing the latter to access variables before they are properly initialized or instantiated. This pattern arises only in RW order violations, where the write is an instantiation operation. Whereas NP order violations occur when a read follows a null assignment, UR violations occur when a read precedes an instantiation operation. To reduce false positives, Themis checks for defensive-programming patterns guarding read accesses.

Missing Retrieval Check (MR). Retrieving an element from a collection or array may return `null`; if the returned value is subsequently used without a null check within the function, `NullPointerException` can occur. An MR bug can originate from either order or atomicity violations. For order violations, MR arises when one access is a write and the other is a read, where the read is a retrieval operation that obtains an element from a collection or array. For atomicity violations, MR arises under the RWR, WWR, and WRW interleavings. In RWR and WWR, the final read is a retrieval operation that returns `null` due to an intermediate write. In WRW, the intermediate read is a retrieval operation and returns `null` only when interleaved between the two writes. Within the function containing the final retrieval operation, Themis checks whether the retrieved element is dereferenced without a preceding null check.

Thread-Unsafe Objects (TO). Concurrent accesses to thread-unsafe collections and IO/NIO objects can lead to bugs. Internal execution interleavings within these library objects may introduce atomicity violations. The resulting symptoms vary based on the internal data structures and implementations [73, 74, 77, 78, 80]. Some failures manifest as explicit runtime exceptions, such as `ConcurrentModificationException` thrown when a collection is iterated while being concurrently modified. Others remain silent, such as lost updates or missed operations, which only surface later when the corrupted state is used. Operations on different entries within the same thread-unsafe collection can still interfere with each other. For example, `HashMap` contains an array where each slot holds a linked list of entries. If `remove()` is invoked concurrently on different entries within the same list, each

Table 7: Eight target systems in the latest stable releases.

System	Release	SLOC	Description
MapReduce	3.4.1	791K	A distributed processing framework.
Yarn	3.4.1	1201K	A distributed resource manager.
HDFS	3.4.1	845K	A distributed file system.
Hadoop	3.4.1	1480K	Foundational libraries for Hadoop.
HBase	2.6.3	1050K	A distributed database.
Alluxio	2.9.6	503K	A virtual distributed storage system.
Hive	4.1.0	2376K	A distributed data warehouse system.
Tez	0.10.5	237K	A distributed data-processing engine.

invocation reads and modifies the list locally before writing it back. Because the operations are unaware of each other, the earlier modification can be overwritten by the later one, leading to a lost update. We manually label thread-unsafe collections and IO/NIO objects, along with their operations, in the standard Java library. If two labeled operations execute concurrently on the same object and at least one operation is a write, Themis reports a TO violation.

A.4 Three Test Generation Prompts

Figure 6 shows the exact prompts used for the generation, the generation with test suites, and the repair, respectively. The placeholders (e.g., public_api_names, source_codes, error_-message) are dynamically instantiated with violation-specific information during query.

A.5 Target Systems and Descriptions of Detected Bugs

Table 7 shows the eight systems used in our evaluation. Table 8 provides short descriptions for each detected bug by Themis, including both the newly discovered and the previously reported ones.

A.6 Detection Time and Token Cost

Themis's static analysis completes in an average of 34.6 minutes across all evaluated systems, with runtime increasing with system size. The majority of the cost is attributed to constructing the program dependency graph (PDG) and call graph (CG).

The LLM-based test generation requires an average of 10.2 minutes per violation, ranging from 18.4 seconds to 43.7 minutes. 212 (93%) violations complete test generation within 15 minutes. The dominant execution cost arises from querying the online LLM, including both network latency and model inference time. Longer generation times mainly result from exploring multiple candidate public interfaces, where each interface pair may undergo up to 10 generation–repair cycles before being discarded. The longest case occurs when a valid test is generated on the 12th attempted public-interface pair.

In terms of token consumption, LLM-based test generation averages 487K tokens per violation, ranging from 103K to 2.1M. 207 (91%) violations consume tokens under 800K.

Table 8: Descriptions of 52 new bugs and 6 known bugs detected by Themis. Six confirmed newly discovered bugs are **bolded**.

#	Bug ID	Description
1	**Alluxio-18738**	**Use the connection before initialization finishes.**
2	Hadoop-19462	Assign unexpected value to the state before reading it.
3	**Hadoop-19458**	**Disable expected mode before mode checking.**
4	HBase-29525	Service is disabled before work finishes.
5	HBase-29516	Close the heap during scanning.
6	HBase-29607	Assigning master to null before dereferencing it.
7	**HBase-29529**	**Access the resources before initialization finishes.**
8	HBase-29530	Access the services before initialization finishes.
9	HBase-29538	Use-after-close causes null dereferencing.
10	HBase-29539	Read the state before initializing it.
11	HBase-29606	Reset the map to empty before retrieval.
12	HBase-29510	Modify store set while iteration.
13	HBase-29595	Modify child list while iteration.
14	HDFS-17726	Assign the property incorrectly before reading it.
15	HDFS-17727	Get connection after it is closed.
16	HDFS-17879	Remove node before registering it.
17	HDFS-17823	Resolve router before initialization.
18	HDFS-17734	Query a work that has been cleared.
19	HDFS-17822	Update the map while adding elements to it.
20	Hive-29143	Release the resource before acquisition.
21	**Hive-29151**	**Access to session states before initializing them.**
22	Hive-29156	Access external resources before initializing them.
23	Hive-29150	Query a task that has been removed.
24	MapReduce-7498	Close cluster before entering the correct state.
25	MapReduce-7501	Create a non-existing segment after reinitialization.
26	MapReduce-7514	Fetch a state that has been cleared.
27	MapReduce-7511	Clear all ACLs while checking the permission.
28	MapReduce-7516	Set server property to null before startup.
29	MapReduce-7529	Access the client while creating the connection.
30	MapReduce-7509	Query the array after it is empty.
31	MapReduce-7499	Access the server before initialization finishes.
32	MapReduce-7510	Access the server before initialization finishes.
33	MapReduce-7515	Shutdown the server before initialization finishes.
34	**MapReduce-7507**	**Double remove the same task after checking its existence.**
35	MapReduce-7518	Use-after-clear race on membership.
36	MapReduce-7512	Concurrent updates to the same counter map.
37	MapReduce-7513	Access a linked hash map while being iterated.
38	Tez-4644	Access an attempt after it has been unregistered.
39	**Tez-4645**	**Context is read before initialization finishes.**
40	Yarn-11769	Calls timeline API after disabling it.
41	Yarn-11770	Resolve the object after it enters an illegal state.
42	Yarn-11772	Reloads service before all setup completes.
43	Yarn-11773	Checks GPU properties after resetting them to empty.
44	Yarn-11771	Get groups after setting them to null.
45	Yarn-11847	Get target user after setting it to null.
46	Yarn-11846	Use federation property before initialization finishes.
47	Yarn-11849	Access jobs before they get into the ready state.
48	Yarn-11852	Use admin service before initialization finishes.
49	Yarn-11853	Use router service before initialization finishes.
50	Yarn-11860	Query an entry that has been removed.
51	Yarn-11851	Clear field map while being iterated.
52	Yarn-11859	Concurrent copy and write to the same file.
53	HBase-4539	Delete a node after it has been deleted.
54	HBase-5780	Start up the node before authentication.
55	HBase-6070	Assign unexpected value to the region before reading it.
56	HBase-4729	Split the region while deleting it.
57	MapReduce-4099	Kill the server before removing all its resources.
58	MapReduce-4637	Kill the task before it is available.

Token usage mainly results from repeated generation–repair cycles and the substantial contextual information supplied during repair. Trying additional public interfaces further increases the cumulative token cost dramatically.

Staged parameter fuzzing and interleaving exploration require an average of 6.1 minutes per violation, ranging from 7.9 seconds to 28.1 minutes. This phase benefits significantly from initial seed pool improvement (constants and LLM-generated parameter values), which improves the quality of initial seeds and reduces the search effort of directed fuzzing.

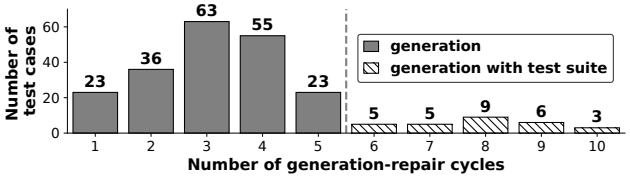
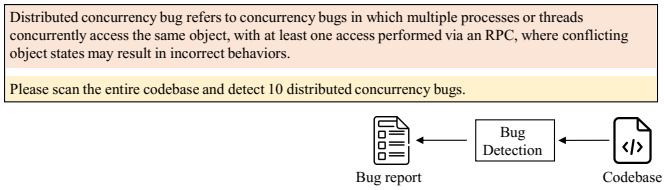
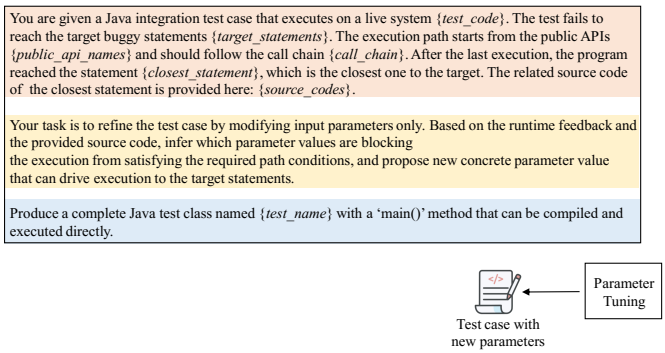


Figure 7: Distribution of generation–repair cycles for generating the executable test case with the appropriate public interfaces in LLM-based holistic test generation. Themis successfully generates 228 error-free test cases in total.



(a) The prompt of LLM-only bug detection in Appendix A.8.1.



(b) The prompt of LLM-only parameter tuning in Appendix A.8.3.

Figure 8: The prompt of the LLM-only study.

For instance, in Figure 1c, a conventional directed fuzzer fails to trigger the violation within the same time budget as Themis (i.e., 30 minutes) because it never mutates the parameter `tState` to the required constant `SUCCESS_FINISHING`. In contrast, Themis extracts this constant from the execution path and inserts it into the initial seed pool, enabling the fuzzer to reach the targets within 2.4 minutes.

A.7 More Results of Test Generation

Figure 7 shows the distribution of required generation–repair cycles for generating the executable test case with the appropriate public interface. Most violations still require more than two generation–repair cycles to yield an executable test, since LLM-generated initial test cases vary in quality. To avoid fixating on low-quality initial test cases, we regenerate a fresh test, increasing the likelihood of restarting from a higher-quality draft that can be successfully repaired to execute without runtime errors. 200 of 228 test cases are produced by the generation without test suites. For the remaining 28 cases, the LLM leverages existing test suites to construct necessary call sequences. For example, in Yarn-11853 bug, the LLM fails to produce a valid call sequence for constructing a router instance and invoking the public interface on it; when supplied with test cases from `TestRouterAdmin.java`, the LLM learns to compose the router instance and register its required services, successfully generating a test case without execution errors.

A.8 Details of LLM-only and Prompt Variants Study

A.8.1 LLM-only Bug Detection without Static Analysis Guidance

We evaluate two LLM-only settings: a zero-shot prompt with only a definition of distributed concurrency bugs, and a few-shot prompt augmented with six real-world bug examples. For the zero-shot setting, the prompt is shown in Figure 8a. For the few-shot setting, we add the bug description, racy statements, the symptom statement, and the whole buggy execution path of the six known bugs to the prompt. Because validating whether a reported violation corresponds to a real distributed concurrency bug requires substantial manual inspection, we randomly select one system and we limit the LLM to output at most 10 candidate violations.

Under the zero-shot and few-shot configurations, all reported candidates are false positives (i.e., none of them correspond to real distributed concurrency bugs), and the LLM fails to identify a single bug detected by Themis. Under the zero-shot and few-shot configurations, the execution time is 5.3 and 5.7 minutes, respectively, while the token cost is 1.7M and 1.9M tokens.

Manual inspection reveals two dominant sources of false positives. First, despite explicitly requiring RPC involvement for distributed concurrency bug detection, most reported violations omit any RPC components, indicating that the LLM frequently misidentifies or ignores distributed interactions. Second, many reported violations are benign races that cannot lead to observable symptoms. Overall, these results show that LLM-only bug detection lacks the precision and semantic grounding required for distributed concurrency analysis, and they justify the necessity of Themis's static-analysis–guided, multi-phase design.

A.8.2 Prompt Variation of LLM-based Test Generation

We explore prompt variations along two orthogonal dimensions: instruction and context. On the instruction side, we evaluate the impact of removing structured guidance and instead of asking the LLM to directly generate a test case. On the context side, we vary the information provided to the LLM, ranging from the curated context used by Themis to richer but less focused alternatives, such as the entire codebase or existing test suites. Due to the high token cost of test generation, we randomly select 10 violations for this study.

Removing structured instructions. In this experiment, we retain the same contextual information as in Themis's default setting, but omit the instruction that explicitly specifies the expected test structure. Instead, we prompt the LLM to directly generate a test case. This prompt removes the second paragraph (i.e., the task part in Figure 6) of the generation prompt. The results show that the LLM generates only one valid test case out of the 10 violations. This successful case takes 3.2 minutes and consumes 162K tokens, and it corresponds to the scenario where the test simply invokes the public-interface pair concurrently without any preceding call sequence. All remaining cases fail after exploring all candidate public interfaces and exhausting the generation–repair cycle budget. In failed cases, the LLM fails to recognize the need to initialize RPC-related components or to construct application-specific call sequences for establishing the required system state, underscoring the importance of the structured instructions in Themis's generation prompt, which guide the model to generate the test case with explicit instructions.

Providing the codebase as context. In this experiment, we provide the LLM with the entire codebase of the target system and ask it to autonomously identify and extract the information needed to construct a test. The instruction remains unchanged, but the available context used by Themis is replaced with the full repository. The prompt in this experiment is a variant of the generation prompt without test suites, with the last sentence of the first paragraph (i.e., the knowledge part in Figure 6) replaced with "You are provided with the entire codebase, and you can extract the code related to the violation for generating the test." The result shows that the LLM successfully generates valid test cases for nine out of the 10 violations. However, this improvement comes at a substantial cost: the approach consumes an average of 17.1 minutes per violation, compared to 9.9 minutes under Themis's default setting, and incurs a token cost of 1.8M per violation, which is approximately four times that of Themis.

Providing the test suite as context. In this setting, we augment the prompt by incorporating existing test suites, specifically those associated with the classes containing the public interfaces. We use the same generation prompt with test suites as in Themis. The result shows that the LLM successfully generates valid test cases for seven out of the 10 violations. The average generation time is approximately 12.9 minutes, compared to 10.5 minutes under Themis's default setting, and the average token consumption is 624K tokens per violation, representing a 28.1% increase. The failures primarily arise because test suites often depend on test-only helper components and mocking, which can mislead the LLM.

A.8.3 LLM-only Parameter Tuning with Runtime Feedback

In this experiment, the LLM operates in a closed feedback loop for parameter tuning. We instrument the system, similar to the fuzzing module in Section 5, to monitor test execution. After each test execution, we identify the basic block closest to the target based on the directed-fuzzing distance metric, and supply the LLM with a focused context consisting of the source code of the basic block's enclosing function, all functions invoked within the block, and the objects accessed in the block. The prompt is shown in Figure 8b. The LLM then modifies the input parameters, and the updated test is re-executed. This loop continues until the target is reached or the search stagnates, i.e., the minimum distance to the target does not decrease for 10 consecutive iterations.

The prompt is shown in Figure 8b. Due to the high token cost, we randomly select 30 violations that need parameter tuning for evaluation. Among these 30 violations, the LLM successfully identifies valid parameters for only six cases. This limited success also incurs a substantial cost: the approach consumes an average of 17.5 minutes, compared to 7.9 minutes under Themis's parameter fuzzing, and 324K tokens per violation, making it significantly more expensive than directed fuzzing.