Fuzzing Enterprise-Grade Blockchain Systems: Industrial Practice and Solutions

Fuchen Ma
BNRist, Tsinghua University

Yuanliang Chen*
BNRist, Tsinghua University

Zhen Yan
BNRist, Tsinghua University

Yuanhang Zhou
BNRist, Tsinghua University

Yu Jiang*
BNRist, Tsinghua University

Mingchao Wan
Beijing Academy of Blockchain and Edge Computing

Abstract

Blockchain has been widely adopted across diverse sectors. Yet, enterprise-grade systems remain vulnerable to critical flaws that undermine stability and security. Although academic fuzzing tools such as LOKI and Tyr have shown effectiveness in detecting such issues, their integration into industrial practice remains challenging.

In this paper, we present the industry practice of implementing system-level fuzzing techniques on enterprise-level blockchains. We summarize three main obstacles in industry deployment, namely the hard-to-build state models for fuzzing, the slow convergence of fuzz testing within CI/CD pipelines, and the difficulty of adapting logical bug oracles across diverse blockchain implementations. To address these obstacles, we design Thor, a practical fuzzing framework for industry blockchain systems. Thor uses active and passive packet generation for early stage state-aware testing. To perform efficient fuzzing under strict CI/CD time budgets, Thor adopts a two-tier parallel fuzzing method. And Thor also uses LLM-based oracles to extract logic properties from node logs. Over these years, Thor has discovered 87 bugs in 9 commercial blockchain systems, such as Chainmaker, Go-Ethereum, and WeBank FISCO BCOS.

CCS Concepts: • **Security and privacy → Distributed systems security**.

Keywords: Fuzz Testing, Blockchain System

ACM Reference Format:
Fuchen Ma, Yuanliang Chen, Zhen Yan, Yuanhang Zhou, Yu Jiang, and Mingchao Wan. 2026. Fuzzing Enterprise-Grade Blockchain Systems: Industrial Practice and Solutions. In *21st European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3767295.3803582

*Yuanliang Chen and Yu Jiang are the corresponding authors.

1 Introduction

Enterprise-grade blockchains typically exhibit: (i) frequent code commits (e.g., 1–8 per day in Aptos), (ii) strict CI/CD time budgets (e.g., 150 minutes in FISCO), and (iii) support for multiple consensus protocols (e.g., Raft and SmartBFT in Fabric). However, the rapid evolution of enterprise-grade blockchain code makes these systems highly error-prone. Such bugs may trigger node crashes or consensus failures, leading to property loss, privacy breaches, or identity theft with severe impact on both individuals and organizations. It is therefore crucial to develop effective methods for detecting blockchain vulnerabilities.

To this end, both academia and industry have developed various blockchain testing tools [1, 18, 20]. Among them, node message fuzzing, as in LOKI [28] and Tyr [7], has shown strong potential. This approach simulates Byzantine attacks by injecting malicious network packets to assess system resilience, while leveraging runtime state awareness to guide input generation and defining rich logic oracles to capture subtle semantic bugs. These tools have uncovered numerous vulnerabilities across blockchain platforms. Nevertheless, applying them to enterprise-grade blockchains still faces major obstacles: (1) frequent state-model staleness, (2) strict CI/CD time constraints, and (3) diverse log/oracle formats.

The first challenge is building effective early-stage state models, especially since in enterprise blockchains these models rapidly become stale, but are still critical for guiding fuzzing. 1) In the development process, code commits frequently alter consensus states or transition rules, rendering state models non-reusable across versions. For example, in FISCO BCOS, 41.7% [17] of the commits in August 2025 modified consensus or scheduling modules, directly impacting fuzzing state modeling. Constructing accurate state models from scratch for complex blockchain systems is inherently slow and imprecise, particularly in the early stages when system behavior is not yet well understood. This in turn delays the discovery of critical bugs and undermines the practicality of existing tools in fast-paced industrial development environments.

The second obstacle is the limited efficiency of fuzzing under strict CI/CD budgets, where rapid convergence is essential. In enterprise environments, blockchain systems are continuously updated and tested through CI/CD pipelines, with each commit potentially introducing system-level vulnerabilities. Ensuring security and stability at the earliest stage is therefore critical. While fuzzing could address this need, its effectiveness is constrained by performance. For instance, according to statistics reported by Tyr, achieving converged code coverage and exposing deep logic bugs can take up to 12 hours. Such a prolonged feedback cycle is incompatible with the short time budgets of industrial CI/CD workflows, where developers expect results within a relatively short time.

The third obstacle is the difficulty of adapting logic bug oracles across heterogeneous blockchain platforms, where log formats differ. Unlike memory corruption bugs, logic bugs in blockchain systems are highly platform-specific and often require deep domain expertise to detect. Existing fuzzing tools, such as LOKI and Tyr, define custom oracles to capture semantic violations, including liveness, safety, integrity, and fairness. However, these oracles are tightly coupled with the implementation details of a given blockchain, making them hard to generalize or reuse. From a tester's perspective, oracle construction itself is a major barrier: it involves manually identifying system invariants, encoding semantic checks, and adapting to diverse log formats across platforms, all of which demand significant effort and specialized knowledge. As a result, every new blockchain system requires substantial re-engineering of oracles, severely limiting the scalability of fuzzing and hindering its broader adoption in industry.

In this paper, we analyze the obstacles to applying effective and general fuzzing in enterprise-level blockchain systems and present Thor, a practical framework that addresses them with three key designs. First, to mitigate state-model staleness, Thor leverages nodes' rich runtime states and adopts a hybrid mode combining active and passive message sending for early-stage, state-aware testing. Second, to cope with strict CI/CD time budgets, it employs a two-tier parallelism strategy across protocol states and message structures, accelerating convergence. Third, to handle heterogeneous log formats, Thor uses Large Language Models (LLMs) to extract oracle-relevant entries from logs and synthesize assertions, enabling adaptable logic oracles across diverse platforms. By working with the engineers and developers in the fuzzing process, we have totally uncovered 87 serious bugs in 9 enterprise-level blockchains such as WeBank FISCO BCOS [26], Chainmaker [6], and Go-Ethereum [13]. 17 bugs are assigned with CVE identifiers due to their severe impacts.

In summary, we make the following contributions:

1. We summarized and analyzed three main obstacles in implementing effective and general fuzzing for enterprise-level blockchain systems.
2. We propose and implement an industrial blockchain fuzzing framework called Thor, involving active and passive message sending, two-tiered parallel fuzzing, and LLM-based logic oracle adoption.
3. We deployed Thor [1]to continuously fuzz 9 enterprise-level blockchains, and identified 87 previously unknown bugs, enhancing their security and earning recognition from the respective vendors.

2 Background

State-Aware Blockchain Fuzzing. As a complex distributed system, a blockchain maintains dynamic and evolving runtime states, including data storage, consensus status, and network topology. Accurately modeling these states is essential for effective system testing, as it enables the generation of semantically meaningful test cases and the detection of deep logic bugs. A state model abstracts key components and their transitions, providing a structured view of the system's behavior over time. However, due to the decentralized and event-driven nature of blockchain systems, constructing such models is inherently challenging and often requires runtime introspection or domain-specific knowledge.

Existing tools like LOKI and Tyr construct a dynamic state model by masquerading as a legitimate node within the blockchain network. It listens to and decrypts real-time network messages, extracting key information such as message types, sender and receiver identifiers, and timestamps. These messages are organized into sequences based on their types and node identifiers. The tools then mine high-frequency patterns from these sequences to identify common state transitions, forming a state model represented as a tree structure. Each node in this tree denotes a specific consensus phase, and edges represent possible transitions between these phases. This model is continuously updated during the fuzzing process, allowing the fuzzer to adapt its input generation strategy based on the evolving state of the system. However, in the early stages of fuzzing, these state models are often incomplete and inaccurate, leading to low testing effectiveness.

Blockchain Properties. Robust blockchain systems are characterized by four fundamental properties: *liveness*, *safety*, *fairness*, and *integrity*, which together define the expected correctness and security of the system. Liveness [25] ensures that the system continues to make progress. Valid transactions submitted to the network will eventually be confirmed and included in the blockchain, preventing indefinite delays or deadlocks. Safety [25] guarantees consistency among honest nodes. Once a block is confirmed, it cannot be reverted or replaced, and all honest nodes should maintain a consistent view of the ledger, preventing issues such as double-spending or fork divergence. Fairness [3] requires that all participants have a fair chance to propose or validate blocks. This property prevents centralized control or censorship by any single entity, and reflects whether the system

[1]THOR is available at: https://github.com/THU-WingTecher/Thor

respects decentralized participation under the intended consensus protocol. Integrity [7] ensures the correctness and immutability of on-chain data. Once transactions are committed, they must not be altered or forged, preserving the auditability and trustworthiness of the blockchain state.

3 Obstacles in Enterprise-level Blockchain Fuzzing

Though the internal designs of enterprise blockchain systems vary, the fundamental steps for enabling continuous fuzzing remain consistent. Figure 1 illustrates three essential stages in the fuzzing workflow: (1) *State-Aware Input Generation*, which leverages runtime system states to construct semantically meaningful testing inputs; (2) *CI/CD Fuzzing*, which integrates fuzzing into continuous development pipelines to detect bugs introduced during frequent code updates; and (3) *Logic Bug Monitoring*, which analyzes execution logs to identify violations of critical system properties such as safety, liveness, and fairness.

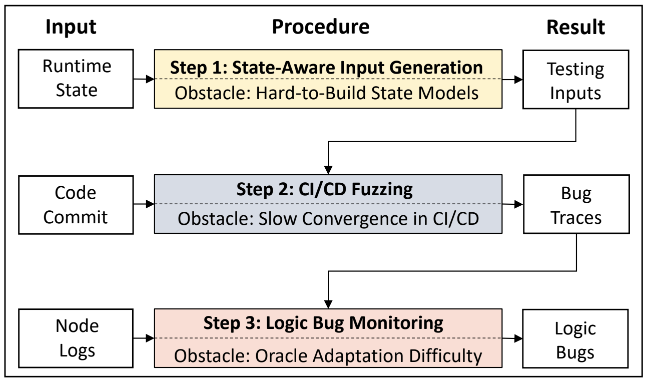

Input
Procedure
Result
Runtime State
Step 1: State-Aware Input Generation
Obstacle: Hard-to-Build State Models
Testing Inputs
Code Commit
Step 2: CI/CD Fuzzing
Obstacle: Slow Convergence in CI/CD
Bug Traces
Node Logs
Step 3: Logic Bug Monitoring
Obstacle: Oracle Adaptation Difficulty
Logic Bugs

Figure 1. Basic steps for conducting general and effective fuzzing on enterprise-level blockchain systems. 1) Generate state-aware testing inputs; 2) Perform fuzzing in the CI/CD pipelines; 3) Monitor the logic bugs at runtime.

3.1 Step 1: State-Aware Input Generation

The first step is to generate fuzzing inputs that align with the internal runtime state of the blockchain system. Unlike traditional input generation, state-aware input generation leverages the dynamic behavior and phase transitions of blockchain consensus protocols to craft valid and effective test messages. This is crucial because many logic bugs in blockchain systems are only triggered under specific states, such as during leader election, vote aggregation, or view changes. Without awareness of the current state, generated inputs may be rejected or ignored, significantly reducing test coverage and the ability to discover deep semantic flaws. Therefore, understanding and incorporating system state into the input generation is essential for effective fuzzing.

Obstacles: The main obstacle in this step lies in the difficulty of constructing an accurate and useful state model, especially in the early stages of fuzzing. Our study of FISCO BCOS shows that 41.7% of commits in August, 2025 [17] modified consensus or scheduling modules, directly disrupting fuzzing state models. This recurring invalidation imposes significant maintenance costs, limiting reusability of the state models across versions. While blockchain nodes maintain rich internal states, extracting and organizing them into a structured model is challenging due to the decentralized and event-driven nature of blockchain systems. Without a reliable state model, fuzzers struggle to generate inputs that match the system's current phase, leading to high rejection rates and low bug exposure. In addition, the construction of state models is typically slow and imprecise at the beginning of the fuzzing process, as it relies on runtime observation and message pattern mining. This results in poor initial performance and delays in reaching effective fuzzing coverage.

3.2 Step 2: CI/CD Fuzzing

The second step is to perform fuzzing within the CI/CD (Continuous Integration/Continuous Deployment) pipeline, where new code commits are frequently integrated and tested. In modern enterprise blockchain systems, each update may introduce subtle logic bugs that compromise security or correctness. Integrating fuzzing into CI workflows allows bugs to be caught early before they reach production environments. Compared to traditional standalone fuzzing, CI/CD fuzzing must operate under limited time constraints and provide fast feedback. Therefore, the fuzzing process needs to be efficient and effective.

Obstacles: While long-term fuzzing is undeniably valuable for uncovering deep vulnerabilities, ensuring efficient short-term fuzzing is equally essential for CI/CD integration considering the strict CI/CD budget. Balancing these two modes remains a practical barrier. Existing fuzzers require a long time to converge. For instance, Tyr reports that it can take up to 12 hours to reach sufficient code coverage and expose deep logic bugs, which is far beyond the acceptable latency in CI pipelines.

3.3 Step 3: Logic Bug Monitoring

The third step is to monitor for logic bugs during fuzzing by analyzing execution behavior and detecting violations of critical system properties. Unlike memory corruption or crash bugs, logic bugs are often subtle and protocol-specific, such as violations of safety, liveness, or fairness. Detecting these issues requires carefully designed oracles that can interpret runtime logs or state changes and determine whether the system has behaved correctly. Effective logic bug monitoring is essential for uncovering deep semantic vulnerabilities that may not manifest as crashes or obvious anomalies.

Obstacles: The main challenge in this step is adapting or designing logic oracles across diverse consensus protocols

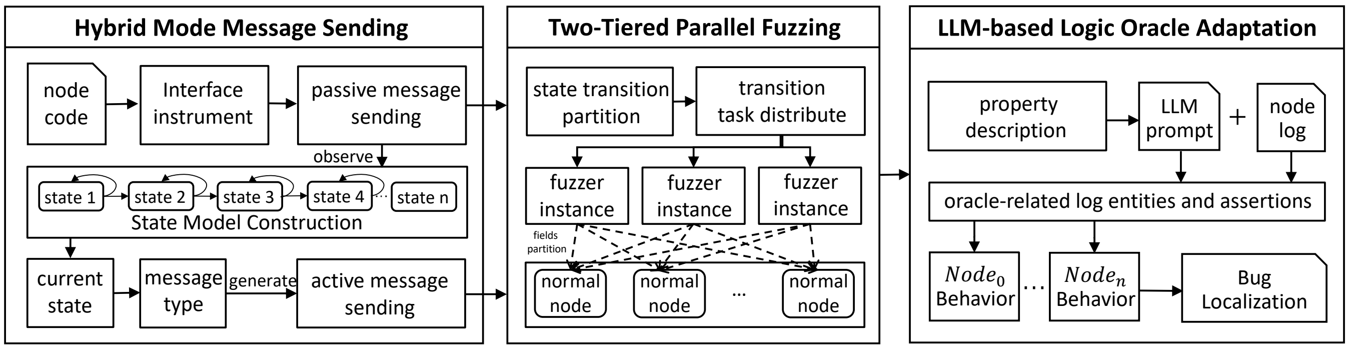


Figure 2. Overview of Thor's design. (1) Thor leverages blockchain state models with a hybrid message-sending strategy (passive + active). (2) It partitions state models and field values to enable parallel fuzzing in CI/CD pipelines. (3) For logic bug detection, Thor applies LLM-based oracle adaptation, generating runtime assertions from property descriptions and logs.

and blockchain systems. From a tester's perspective, oracle construction is a major barrier: it involves manually identifying invariants, encoding semantic checks, and handling heterogeneous log formats. Because different protocols maintain distinct state representations and customized logging, oracles developed for one system are often incompatible with others, requiring significant manual re-engineering. This lack of reusability greatly limits the scalability of fuzzing frameworks, and without accurate oracles, many logic bugs remain undetected even when triggered during fuzzing.

4 Solution of THOR

To tackle the above mentioned obstacles, we designed Thor, as shown in Figure 2. Thor adopts three main components to generate state-aware testing inputs, perform efficient CI/CD fuzzing and monitor logic bugs. (1) Thor leverages blockchain state models with a hybrid message-sending strategy (passive + active). (2) It partitions state models and field values to enable parallel fuzzing in CI/CD pipelines. (3) For logic bug detection, Thor applies LLM-based oracle adaptation, generating runtime assertions from property descriptions and logs.

Thor targets the correctness conditions of blockchain systems through two core mechanisms: 1) Message mutation: By mutating consensus messages (e.g., view-change, roundQC), Thor explores behaviors that may compromise liveness or fairness. 2) Runtime oracles over logs: Thor leverages LLM-based semantic oracles that check for violations of correctness conditions in execution traces. Liveness violations are detected by monitoring stalled transaction processing, integrity violations by inconsistent block heights, and fairness violations by biased transaction ordering. In Thor, the fuzzing nodes are not independent side processes, but masquerading nodes embedded directly into the consensus quorum. They behave as fully participating replicas, capable of sending and receiving consensus messages, so that quorum-sensitive properties (e.g., missing a view-change proposal quorum) can be violated and detected.

4.1 Hybrid Mode Message Sending

In this section, we will give the detailed design of the passive and active message sending modes.

Passive message sending. Figure 3 illustrates the workflow of passive message sending in Thor, which simultaneously serves two purposes: generating testing packets and constructing the state model. When a blockchain node receives an incoming message, it invokes the corresponding handler to process the message and determine which response message should be sent. Thor hooks into this process before the outgoing message is finalized and serialized. As shown by the red arrows, it intercepts the ongoing packet and applies field-level mutations, transforming it into a testing packet that is still syntactically valid and compatible with the protocol logic. This enables the tool to inject test cases into the normal execution path without disrupting the consensus flow.

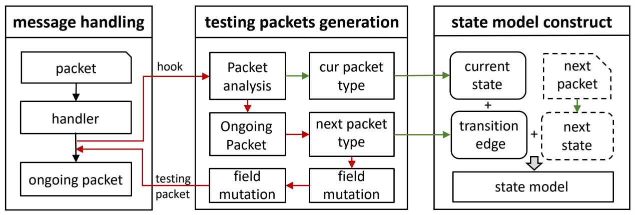


Figure 3. Overview of passive message sending in Thor. The red arrows illustrate how the system hooks into the message handling process to mutate them before they are sent. The green arrows show how Thor records message types and transitions to construct a passive state model incrementally.

At the same time, as highlighted by the green arrows, Thor extracts the current consensus state and infers a transition edge based on the type of the outgoing message. Specifically, it treats the current state as the source node in the state model, and the outgoing message type as the symbolic label of the transition. However, the target state of this transition cannot be fully determined until the next incoming message is observed. To address this, Thor temporarily marks the target state as unknown and records the transition as a pending edge. When the next message arrives, the tool resolves the edge by mapping it to the appropriate destination state, thereby completing the state transition. By repeating this process throughout protocol execution, Thor incrementally constructs a passive state model that captures both common and rarely exercised transitions in the blockchain protocol. This model not only reflects real-world execution paths but also enables state-aware active fuzzing in later stages.

Active message sending. While constructing the state model during passive execution, Thor enters the active fuzzing phase, where it systematically explores protocol behaviors by injecting mutated messages based on the learned transitions. For each round of fuzzing, Thor selects an outgoing message type according to the state model's edges from the current consensus state. Instead of constructing new messages from scratch, it retrieves previously observed messages of the same type (collected during passive fuzzing) and applies field-level mutations to create new test inputs. This reuse-and-mutate strategy ensures that the generated messages are structurally valid and semantically plausible, while still exploring untested execution paths. By aligning message generation with the protocol's actual runtime state, Thor significantly increases the likelihood of triggering deep semantic bugs and avoids sending inputs that would be rejected immediately by the system.

4.2 Two-Tiered Parallel Fuzzing

To maximize fuzzing efficiency in CI/CD pipelines, Thor adopts a two-tiered parallelism strategy that operates across both the protocol state space and the message structure. As shown in Figure 4, each fuzzing round is decomposed into two parallel tasks: at the first level, different state transitions from the current state are explored concurrently by separate fuzzing instances; at the second level, each instance performs message-level parallelism by mutating different fields and dispatching them to multiple target nodes. This hybrid parallelism design enables broad protocol coverage and fine-grained semantic exploration simultaneously, ensuring both depth and diversity in fuzzing campaigns.

State transition parallelism: To accelerate coverage across the protocol state space, Thor implements state transition parallel fuzzing. Given a current system state, there may exist multiple outgoing transitions corresponding to different message types or protocol branches. As shown on the left side of Figure 4, each of these transitions is assigned to a separate fuzzer instance, which independently executes targeted fuzzing campaigns to trigger the corresponding state changes. To achieve this, Thor schedules different fuzzer instances to generate testing messages of different types.

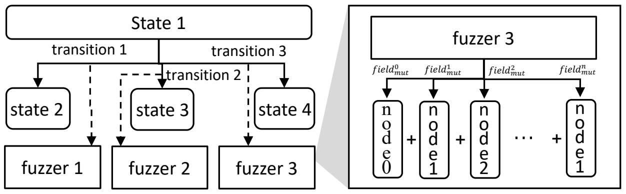


Figure 4. Parallel fuzzing in Thor across both state transitions and message fields. Each transition from the current state is assigned to a dedicated fuzzer instance. Within each instance, multiple message variants are created by mutating individual fields and distributed to different target nodes, enabling broad semantic exploration in parallel.

For example, if the current state is the `Preprepare` in FISCO BCOS, and it has four transitions, Thor will tell the fuzzer instance 1 to generate a testing packet with the `Prepare` type, which tries to transit the state along with the transition edge 1. Similarly, Thor will tell the second fuzzer instance to generate a testing packet with `Viewchange` type and transits as indicated by the second transition edge. This design allows the system to explore divergent protocol paths simultaneously, trying to reduce the code coverage convergence time and improve transition coverage in dynamic blockchain's CI/CD environments.

Message field parallelism: In addition to state-level parallelism, Thor employs fine-grained parallelism at the message field level to explore diverse execution behaviors within a single transition. As illustrated on the right side of Figure4, each fuzzer instance mutates different fields of a testing message independently, such as numeric counters, view numbers, timestamps, or cryptographic hashes, based on their semantic roles and protocol constraints. Importantly, these mutations are structure-aware: they preserve the syntactic and type-level validity of each field while modifying its value to explore edge cases and alternative logic paths. For example, a numeric field may be incremented, negated, or replaced with a boundary value, while a hash field may be subtly corrupted to mimic signature mismatch scenarios.

Parallelism is achieved by deploying multiple fuzzing nodes in the same testnet, with each node exploring a different mutation path. The degree of parallelism is naturally bounded by the tolerated fraction of faulty nodes in the underlying consensus protocol (e.g., up to f faulty replicas in a 3f+1 system). This setup ensures that fuzzing remains realistic while still covering diverse behaviors in parallel.

4.3 LLM-based Logic Oracle Adaptation

Existing blockchain fuzzing tools such as LOKI and Tyr propose manually designed logic monitors to detect semantic bugs related to properties like liveness, safety, and fairness.

While effective, these monitors are tightly coupled with specific protocol implementations and require substantial manual effort to adapt when targeting new blockchain systems. To address this limitation, Thor introduces an automated logic oracle adaptation framework based on large language models (LLMs). By combining a high-level description of the desired property with execution logs collected from multiple nodes, our system queries the LLM to synthesize protocol-specific assertions that can be applied across platforms with minimal human intervention.

Algorithm 1: LLM-based Logic Oracle Adaptation

```
   Input  : $P$: Property description (e.g., liveness, safety)
            $L = \{log_0, ..., log_n\}$: Logs from $n$ nodes
   Output : Bug report if any node violates $P$
1  $prompt \leftarrow$ generatePrompt($P$, $L$);
2  $response \leftarrow$ queryLLM($prompt$);
3  $entities \leftarrow$ extractEntities($response$);
4  $globalView \leftarrow$ [ ];
5  foreach $log_i \in L$ do
6      $e_i \leftarrow$ keyEntities($log_i$, $entities$);
7      $globalView$.append($e_i$);
8  end
9  $assertions \leftarrow$ extractAssertion($response$);
10 $check \leftarrow$ applyAssertions($globalView$, $assertions$);
11 if $check$ = fail then
12     return Bug report: violation of $P$;
13 end
14 return All nodes satisfy $P$;
```

Algorithm 1 shows the workflow of the adaptation process. It first generates a prompt that combines a natural language description of the target property P with historical logs L collected from multiple blockchain nodes (line 1). The prompt is passed to the LLM, which returns a structured response containing two key components: a set of relevant log entities and a set of property-specific assertions (lines 2–3, 7). These log entities capture semantically important elements related to the property (e.g., block commit events, timeout markers, transaction status), and will be extracted from the logs of each node to form a unified view of system behavior.

Next, Thor iterates over each node's logs to extract the relevant entities using the LLM-defined schema (lines 4–6), and aggregates them into a single global view. This global view is then used to evaluate whether the extracted assertions are satisfied collectively (line 8). If any assertion fails when applied to the global view, the system generates a bug report indicating a violation of the given property (lines 9–10). Otherwise, it concludes that the current logs satisfy the specified logic condition.

To better illustrate the oracle adaptation process, we present two concrete examples based on liveness and fairness, which are among the most critical properties in blockchain consensus protocols. 1) For liveness: the LLM first identifies log entities related to finalized blocks and committed transactions across all nodes. It then extracts the number of transactions successfully written to the ledger by parsing confirmation or commit events. The generated assertion compares the total number of committed transactions with the number of client-submitted queries. If the committed transaction count is lower than the total queries issued during the execution window, the system flags a liveness violation, indicating that some transactions were dropped or stalled. 2) For fairness: the LLM analyzes logs to locate events that imply a node is acting as the block proposer. From these, it derives each node's block production count. The assertion checks whether the difference in block counts among nodes exceeds a predefined fairness threshold (as used in Tyr).

```python
m = re.search(r"CommitBlock success.*block=(\d+)", log)
height = int(m.group(1))
if height <= last_height[node]:
    counter[node] += 1
    if counter[node] >= 10: # repeated violation
          raise AssertionError("Liveness violation")
else:
    counter[node] = 0
last_height[node] = height
```

Figure 5. A case of checking scripts generated by the LLM.

THOR prompts the LLM to produce executable scripts that combine (i) regular-expression–based entity extraction rules for parsing logs and (ii) logical assertions for property checking. These assertions are expressed as standard Python assert statements or conditional checks. For example, given the log entry "CommitBlock success, block number=XXX" in FISCO BCOS, the LLM-generated script extracts the block number field via a regex, tracks its progression across nodes, and issues an assertion as shown in Figure 5.

5 Implementation

Based on the above solutions, we implement a continuous fuzzing framework for blockchains, named Thor. It mainly contains three parts: a hybrid message generator, a two-tiered parallel fuzzer, and a LLM-based logic oracle producer.

The hybrid message generator is implemented with a dedicated thread responsible for active message generation. In this thread, Thor monitors the current consensus state of the system and selects appropriate message types to construct testing inputs based on the learned state model. These messages are retrieved from the historical message pool collected during passive execution and are mutated before being dispatched to the target nodes. For passive message generation, we re-implement the packet construction interface inside the blockchain node's codebase. By hooking into the message sending path immediately before serialization, Thor injects field-level mutations into outgoing messages without interfering with the original control flow. Additionally, the tool logs the message type and surrounding protocol context at each send point, which is used to incrementally construct the state transition model during runtime.

As for the two-tiered parallel fuzzer, Thor assigns fuzzing tasks based on the structure of the state model and the number of available fuzzing nodes. Each transition in the state model is assigned a unique index starting from 1. During execution, the i-th testing node is initially responsible for fuzzing the i-th transition from the current state. If the number of outgoing transitions exceeds the number of available fuzzer instances, the system rotates assignments upon the next encounter of the same state, prioritizing transitions that have not been fuzzed yet. This strategy ensures balanced exploration of all possible transitions from each state while minimizing redundant effort. The same indexing strategy is applied at the message field level for field-level parallelism. Non-fuzzer nodes are deterministically assigned based on their IDs to receive mutated messages that differ in specific fields. During deployment, the number of testing nodes is determined according to the system's consensus protocol. For example, in BFT protocols, Thor assigns one-third of the nodes as fuzzers, ensuring that the testing workload does not compromise the consensus model during fuzzing.

For logic oracle adaptation, Thor uses GPT-4o. Once the LLM returns a response, Thor parses the output into a list of entity extraction rules and logical assertions. The logs are then analyzed using these assertions to verify whether the system's behavior satisfies the specified property.

Overall Comparison. We listed the differences between Thor and existing blockchain or protocol fuzzing tools in terms of their functionalities in Table 1. Thor uniquely supports both active and passive message generation, allowing it to explore a wider input space than tools such as LOKI, Tyr, and Peach, which rely solely on synthetic packet injection, or Fluffy, which lacks protocol-level message generation. More importantly, Thor is CI/CD-aware, enabling deployment within real development pipelines to fuzz new commits automatically—something that none of the compared tools currently support. In terms of logic oracle adaptation, Thor is the first to leverage LLMs to automatically synthesize property-specific assertions, while LOKI, Tyr, and Peach require manual adaptation and Fluffy relies on differential testing. Finally, Thor demonstrates broader applicability, supporting 9 blockchain systems in contrast to LOKI (4), Tyr (6), Fluffy (1), and Peach (no native blockchain support).

6 Evaluation

Experiment Setup. All experiments are performed on a machine with 128 cores (AMD EPYC 7742 Processor @ 2.25 GHz) and 128 GB of RAM. The machine is running an Ubuntu 22.04.3 operating system with Linux kernel version 5.15.0. All blockchain systems are deployed with a 10-node cluster, where 3 are fuzzing nodes. We ran each blockchain long-term testing for 24 hours, which is a commonly used time frame for fuzzing evaluation [23].

Table 1. Comparison of Thor with state-of-the-art blockchain or protocol fuzzing tools. Thor uniquely supports both active and passive message generation, is CI/CD-aware, and adapts logic oracles automatically via LLMs.

	Message Generation Methods	CI/CD Fuzzing	Oracle Adaptation	Supported Systems
Thor	Active & Passive	CI/CD-aware	Auto by LLM	9
LOKI	Active	No Support	Manually	4
Tyr	Active	No Support	Manually	6
Fluffy	/	No Support	Differential	1
Peach	Active	No Support	Manually	/

6.1 Effectiveness of Bug Finding

In this section we evaluate whether Thor is effective in finding bugs compared with existing blockchain fuzzing tools. We ran Thor and other tools on a specific version of 3 blockchain systems, including FISCO BCOS [26], Chainmaker [6], and Go-Ethereum [13], for 24 hours and collect the number of bugs they found. It should be mentioned that if we found any bug, we will restart the blockchain cluster to further explore new potential bugs. Specifically, we tested the FISCO BCOS version 3.9.0, Chainmaker version 2.3.4, and Go-Ethereum version 1.10.25. The number of bugs found in 24 hours is shown in Table 2.

Target Selection. To ensure a fair comparison across systems, we selected the set of blockchains that are supported by similar fuzzers under study. The intersection of what Thor, LOKI, and Tyr can fuzz includes FISCO BCOS, Go-Ethereum, and Diem. Among these, we excluded Diem because it has been unmaintained since 2023 (last commit in that year), and several dependent libraries are no longer actively supported, which makes reproducible experiments unreliable. Therefore, our comparison focused on FISCO BCOS and Go-Ethereum, which remain representative, widely deployed enterprise-grade and public blockchains. To further demonstrate generality, we extended the evaluation to Chainmaker, a production-grade consortium blockchain and used in enterprise applications.

As the table shows, in 24 hours, Thor found 41 bugs in 3 blockchain systems, 18-39 more than state-of-the-art tools. The reason why Thor can find more bugs is that it contains a hybrid message generation method, and parallel fuzzing strategy. Specifically, Thor found 27 bugs in the first two hours, while LOKI, Tyr, Fluffy and Peach found 8, 13, 2, and 9. This shows Thor accelerates bug discovery, which is critical for CI/CD pipelines.

Table 2. Number of bugs found by THOR and other tools in 24 hours on specific blockchain versions.

Tool	FISCO BCOS	Chainmaker	Go-Ethereum
THOR	15	15	11
LOKI	3	7	5
Tyr	7	8	8
Fluffy	/	/	2
Peach	2	4	3

Bug Severity. About half of the bugs are due to memory-related issues, such as memory leak, buffer overflow, null pointer dereferences, and use-after-free. Such bugs can lead to blockchain node crashes, and cause significant damage to the entire system's service. Other bugs are logic problems that may affect the transaction consensus or execution process of the entire system. These bugs may lead to transaction failures, or execution hang. As a result, some invalid transactions will be confirmed [36, 39, 42], while some valid transactions are never committed [35, 37, 38]. This may further lead to property losses and business logic failures.

Bug Exploitation. As a decentralized system, industrial blockchain needs to tolerate byzantine faults. Therefore, nodes may engage in malicious behaviors, and the system needs to tolerate such malicious faults. The bugs identified by THOR can be exploited by sending malicious messages from a blockchain node. The memory related bugs can be leveraged to conduct DoS attacks. The attackers can crash down certain nodes by sending well-crafted messages. While the logic bugs can be leveraged to conduct byzantine attacks, such as double-spending [8].

6.2 Effectiveness of Hybrid Message Generation

In this section, we evaluate the effectiveness of the hybrid message generation by implementing a minus version of THOR named THOR$_{act}$, where we delete the passive message generation part. Table 3 shows the branch coverage in 24 hours on three blockchain systems under the testing of THOR and THOR$_{act}$.

Table 3. The branch coverage of THOR and THOR$_{act}$, which is a minus version of THOR without the hybrid message generation strategy.

	FISCO BCOS	Chainmaker	Go-Ethereum
THOR	31,835	12,466	11,683
THOR$_{act}$	29,185	11,209	10,752
Increment	+9.08%	+11.21%	+8.66%

Across the three targets, THOR covers 9.65% more branches than its ablated variant THOR$_{act}$ on average. Concretely, THOR explores 2,650 additional branches on FISCO BCOS, 1,257 extra branches on Chainmaker, and 931 extra branches on Go-Ethereum, which corresponds to relative improvements of +9.08%, +11.21%, and +8.66%, respectively. The gap stems from branches that are guarded by implicit protocol states, which are hardly reachable by purely active fuzzing. THOR's passive mode mutates the in-flight messages produced by the node itself and therefore exercises those hidden paths as soon as they are encountered.

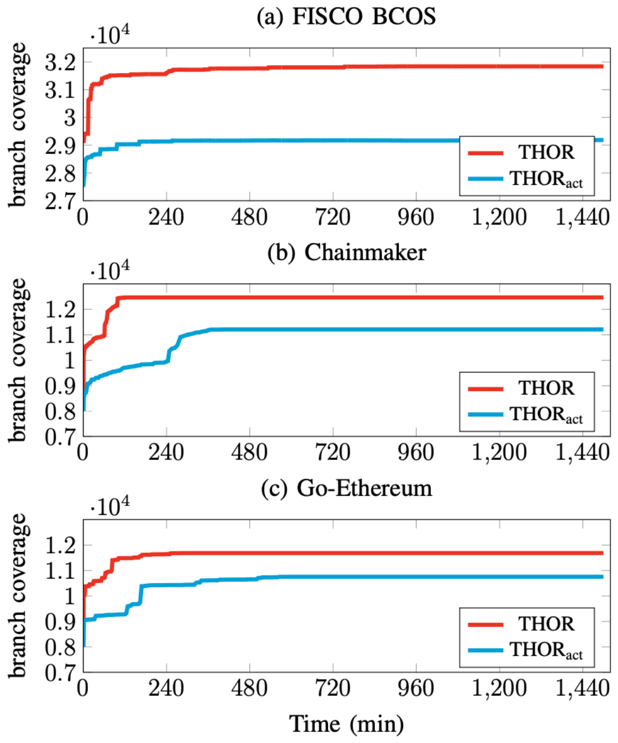


Figure 6. Branch coverage evolution of THOR and its ablated variant THOR$_{act}$ over 24 hours on three blockchain systems. The x-axis shows elapsed time (minutes), and the y-axis shows the number of branches covered by each tool.

Figure 6 plots the coverage curves over time. THOR's line rises steeply during the first hour because it can leverage the node's inherent state transitions immediately; in contrast, THOR$_{act}$ exhibits a long warm-up phase: with no prior state knowledge it spends the early rounds sending syntactically valid but semantically irrelevant packets that are discarded by the peers. Only after enough observations does its active generator begin to approximate the true state graph, at which point the slope of its curve starts to increase. The result confirms that hybrid message generation shortens the exploration warm-up and exposes protocol-specific branches that would otherwise remain hidden.

6.3 Effectiveness of Two-Tiered Parallel Fuzzing

To evaluate the effectiveness of our two-tiered parallel fuzzing strategy, we implement an ablated variant named $\text{THOR}_{\text{nopar}}$, in which all parallelism mechanisms are disabled. In this variant, each fuzzer instance sequentially explores the entire state model and mutates message fields without any distribution across nodes.

Figure 7 presents the branch coverage trends of THOR and $\text{THOR}_{\text{nopar}}$ over a 24-hour fuzzing campaign on representative blockchain systems. We observe that THOR, equipped with two-tiered parallel fuzzing, exhibits a steep increase in branch coverage during the early phase and achieves convergence within approximately 30-100 minutes. After this point, the coverage fluctuates by less than 1.5%, indicating that almost all reachable branches have been explored. In contrast, $\text{THOR}_{\text{nopar}}$ demonstrates slower progress. Its coverage curve continues to rise gradually in about 8 hours, requiring more time to approach a comparable level of exploration.

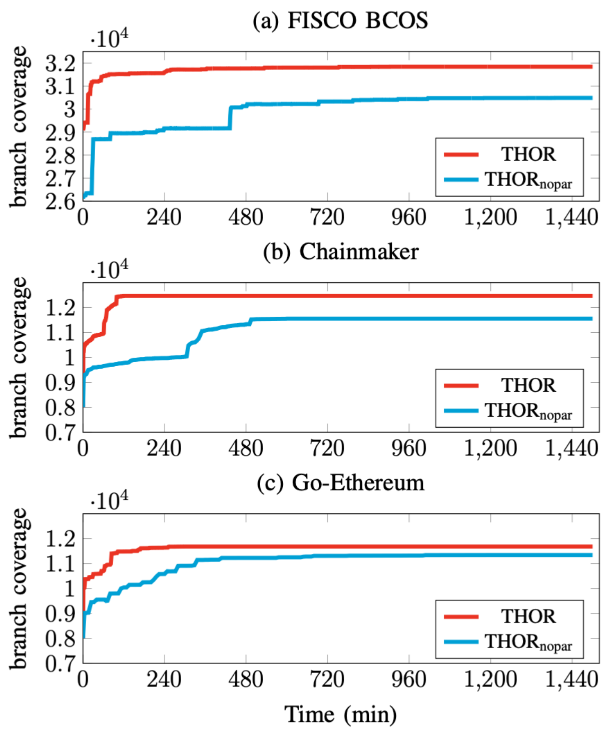


Figure 7. Branch coverage evolution of THOR and its ablated variant $\text{THOR}_{\text{nopar}}$ over 24 hours on three blockchain systems. The x-axis shows elapsed time (minutes), and the y-axis shows the number of branches covered by each tool.

Furthermore, THOR achieves a ultimately higher coverage. This is attributed to the diverse execution behaviors triggered by the two-tiered parallel mode. In parallel settings, multiple nodes simultaneously execute distinct message handling paths, leading to more intricate consensus interactions. These interactions uncover deeper and more complex code branches that are less likely to be reached in the no-parallel fuzzing setup.

These results highlight the practical value of our two-tiered parallel fuzzing design. By ensuring rapid convergence within a tight time budget, THOR enables blockchain developers to obtain meaningful vulnerability detection results early in the testing cycle. Such a property is critical for CI/CD scenarios, where testing time is constrained and rapid feedback is essential for maintaining system security and robustness.

6.4 Accuracy and Robustness of LLM-based Oracles

Accuracy. For false negative evaluation, we tested 15 latest known bugs across 3 platforms. A detailed breakdown of the tested bugs is available in our artifact repository [2] as listed in the following table. Among them, 13 were correctly detected, while 2 were missed (FN rate: 13.3%). The first missed bug is bug#1301[5] in Chainmaker, this is due to that the LLM we used currently cannot match the exact log entities with the config files, without knowing the config items in the config files. The other is the bug#30119[43] in Go-Ethereum, due to that the LLM does not have a whole picture of the DB links between block hash and the block number. Thus, the link missing cannot be extracted automatically by the LLM from the node log. We will try to provide more related files (config files, database index, etc) to the LLM to extract more properties and assertations to eliminate such false negatives. During a 24h continuous testing process, no false positives were observed, since THOR adopts a 6-block decision time window (as suggested by Tyr) to handle transient log inconsistencies.

Robustness. To evaluate the robustness of THOR under incomplete log information, we tested under FISCO log levels (INFO–DEBUG). The oracle correctly detected all logic bugs using stable fields (e.g., height, round, leader), which are explicitly logged, allowing accurate LLM extraction without hallucination. As for the logging inconsistencies across clients, they do not affect reliability, since all major blockchain implementations we tested log these consensus-critical fields in a similar manner.

6.5 Practical Deployment

To demonstrate the practical applicability of our approach, we deployed THOR across 9 mainstream enterprise-grade blockchain systems, covering a diverse set of consensus protocols, execution models, and codebases. The deployments span permissioned blockchains such as Hyperledger Fabric and FISCO BCOS, as well as permissionless systems including Go-Ethereum and EOSIO. Each deployment involved continuous fuzzing over an extended period, integrated into the

[2] The tested bugs are at: https://anonymous.4open.science/r/Thor-E264/LLM-based-Oracle-Evaluation.md

systems' development or testing workflows where possible. This allowed us to evaluate THOR's ability to detect semantic vulnerabilities in realistic, production-grade blockchain environments.

6.5.1 Bugs by Continuous Fuzzing.

Totally, THOR has detected 87 bugs in 9 blockchain systems. Table 4 shows the detailed number of found bugs in each blockchain system.

Table 4. Number of bugs reported by THOR

Blockchain	Tested Version	Reported	Confirmed
FISCO BCOS [26]	2.7.2-3.11.0	25	25
Chainmaker [6]	2.3.4	15	15
Go-Ethereum [13]	1.10.25	11	10
Fabric [21]	2.3	13	13
Quorum [9]	22.7.2	6	6
EOS [52]	2.0.13	2	2
Sei [48]	4.1.7, 5.8.0	10	4
Diem [12]	1.3	4	0
Binance-BSC [2]	1.1.17	1	0
Total	-	87	75

Out of the 87 total bugs, we identified 57 logic bugs, categorized into 38 liveness, 4 fairness, 7 safety, and 8 integrity bugs. We reported all identified bugs to the corresponding blockchain system vendors and received positive feedback. Among the 87 detected bugs, 75 have been confirmed by the maintainers, and 17 have been assigned CVE identifiers and published in the US National Vulnerability Database (NVD).

6.5.2 THOR on Evolving Blockchain Implementations.

We examine THOR's effectiveness on evolving blockchain implementations over a two-year period. Specifically, we analyzed different versions of FISCO BCOS releases from 2021.12 to 2023.12. We collected the bugs found by THOR related to the newly introduced features or code by some versions. The results are summarized in the following table, which shows that THOR consistently identifies newly introduced bugs across versions.

Table 5. Bugs found across evolving FISCO BCOS versions (2021–2023) due to newly added code. THOR consistently discovers newly introduced bugs as the system evolves.

Versions Introducing New-Feature Bugs	Report Date	Bug	Newly Introduced Code or Feature
v3.0.0	2021-12-17	#2079	Code in BlockHeaderImpl has SEGV signals
v3.0.0-rc2	2022-03-01	#2211	Newly added FastViewChange causes hangs
v3.1.0	2022-11-29	#3177	Executor code errorly handles timeout packets
v3.1.1	2022-12-23	#3271	Code in the new DMC feature leads to SEGV
v3.5.0	2023-12-18	#4120	Parallel execution code leads to memory leak

For example, in version 3.0.0-rc2 (March 2022), THOR detected bug #2211 [15], where the newly introduced *FastViewChange* mechanism triggered occasional hangs during consensus transitions. Specifically, the FastViewChange code attempted to accelerate the view change process, but under certain conditions it would block the progress of the commit pipeline. THOR exposed this by constantly mutating the 'view' field in view–change messages to a large value and observing through round and leader log signals, the tool induced a persistent hang (no progress in commits), flagging a liveness violation. Likewise, in version 3.1.1 (December 2022), THOR uncovered bug #3271 [16], a segmentation fault arising from the newly added code in the *Deterministic Multi-Contract* (DMC) parallel execution feature. The DMC scheme attempted to parallelize transaction application across multiple sub-shards to improve throughput, but the implementation had a flaw in proposal handling that could lead to memory issues. By generating messages with conflicting transaction sequences across DMC paths and monitoring for crashes via the log-based oracle, THOR surfaced the memory-safety regression that led to node crashes.

These cases illustrate that THOR not only tracks evolving blockchain implementations effectively, but also systematically exposes subtle regressions introduced by new features, demonstrating its value for continuous testing across software evolution.

6.5.3 Practical case studies.

Case Study 1: A logic bug in FISCO BCOS that stops valid transactions from executing. The related code snippet is shown in Figure 8. This bug is assigned with a CVE id: CVE-2022-28937 [41].

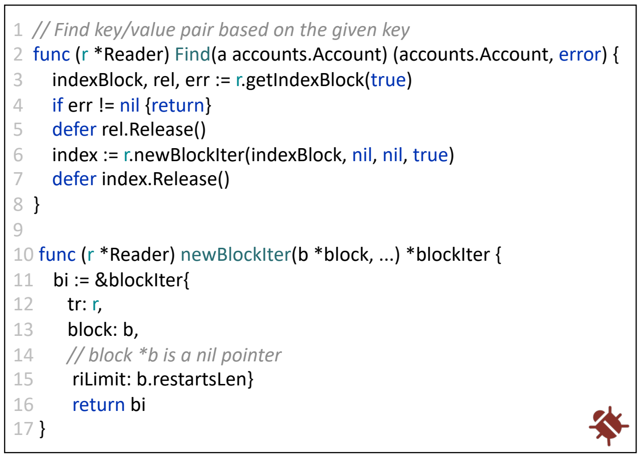
```
void handleMsg(shared_ptr<PBFTBaseMessageInterface> _msg){
+   if (_msg->view() > MaxView){
+     LOG(WARNING)<<LOG_DESC("reject msg with invalid view");
+     return;
+   }
    ...
    handleViewChangeMsg(viewChangeMsg);
}

bool handleViewChangeMsg(ViewChangeMsgInterface::Ptr _msg){
...
leaderInNewViewPeriod(_msg ->index() + 1, _msg ->index());
...
}
```

Figure 8. The logic bug in FISCO BCOS that triggers transactions hang, due to an *INT_MAX* 'view' field.

The bug is caused by the view change process in FISCO BCOS. The fuzzer node becomes a leader and sets the field 'view' in the view change message as the maximum integer (*INT_MAX*). And the other normal nodes try to change the view to $INT\_MAX + 1$, which overflows to 0. And the view 0 indicates that the new leader is the node with index 0, which is still the fuzzer node. Thus, the malicious fuzzer node constantly triggers the view change and all the nodes fail to execute new transactions.

THOR detects this bug through its hybrid message generation strategies. Specifically, during passive message generation, THOR mutates the view' field value based on ongoing messages observed during normal protocol execution. By setting the view' field to *INT_MAX*, the fuzzer node successfully triggers the vulnerability. This case can also be found by LOKI and Tyr in around 200 minutes, while Thor found it in 37 minutes, due to its parallel design. Developers from FISCO BCOS have acknowledged and fixed this issue [11] by introducing an assertion to validate the 'view' field, rejecting any messages with abnormally large values.

Case Study 2: A node crash bug in the block sync process of Go-Ethereum [47]. An attacker can cause the target node to break down by sending a well-crafted getIndexBlock request, thereby compromising the security of the blockchain.

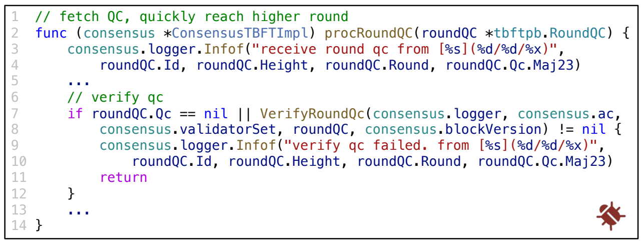


```
// Find key/value pair based on the given key
func (r *Reader) Find(a accounts.Account) (accounts.Account, error) {
  indexBlock, rel, err := r.getIndexBlock(true)
  if err != nil {return}
  defer rel.Release()
  index := r.newBlockIter(indexBlock, nil, nil, true)
  defer index.Release()
}

func (r *Reader) newBlockIter(b *block, ...) *blockIter {
  bi := &blockIter{
    tr: r,
    block: b,
    // block *b is a nil pointer
     riLimit: b.restartsLen}
     return bi
}
```

Figure 9. The node crash bug in the block sync process of Go-Ethereum cased by '-1' IndexBlock request.

Figure 9 illustrates the details of this vulnerability. The function 'find()' is designed to locate a (key, value) pair where the key is greater than or equal to a given key. It first retrieves an IndexBlock through the 'getIndexBl ock()' function using the current reader r, and then calls the 'newBlockIter()' function based on this IndexBlock. The 'newBlockIter()' function is used to create a new block iterator for traversing all (key, value) pairs within a block. However, if the current node receives a getblock request with the *IndexBlock* set to -1, it fails to retrieve a valid block, and the 'getIndexBlock()' function returns a nil pointer. Since there is no nil check in the calling function, the program crashes upon receiving the nil pointer. In this way, the fuzzer node sets the field to -1 and detects the bug. This case can also be found by LOKI and Tyr in around 190 minutes, while Thor found it in 23 minutes. Thor can trigger this by the passive message generation strategy. Thor achieves the semantic information of the field 'IndexBlock', and mutates its value accordingly.

Case Study 3: A system outage bug in Chainmaker [4] caused by an incorrect order of checks during a version update. When a maliciously crafted QCRound request is received, all nodes crash and go offline due to a nil pointer, resulting in an outage of the entire blockchain system.



```
1  // fetch QC, quickly reach higher round
2  func (consensus *ConsensusTBFTImpl) procRoundQC(roundQC *tbftpb.RoundQC) {
3      consensus.logger.Infof("receive round qc from [%s](%d/%d/%x)",
4          roundQC.Id, roundQC.Height, roundQC.Round, roundQC.Qc.Maj23)
5      ...
6      // verify qc
7      if roundQC.Qc == nil || VerifyRoundQc(consensus.logger, consensus.ac,
8          consensus.validatorSet, roundQC, consensus.blockVersion) != nil {
9          consensus.logger.Infof("verify qc failed. from [%s](%d/%d/%x)",
10             roundQC.Id, roundQC.Height, roundQC.Round, roundQC.Qc.Maj23)
11         return
12     }
13     ...
14 }
```

Figure 10. The system outage bug in Chainmaker introduced by a version update of TBFT v2.3.4.

Figure 10 shows the related code that triggered this bug. In Chainmaker, when a node detects that its block height is lower than that of other nodes, it uses the procRoundQC function to catch up with the block round and maintain consistency. Under normal circumstances, when a roundQC request is received, a series of validity checks are performed on the request, as shown in lines 6-13. However, during a version update, a log info statement was introduced, as shown in lines 2-4. This logging logic was incorrectly placed between the QC validity checks, causing the nil pointer issue in Go when the incoming roundQC.QC is nil, leading to the node crashing. This bug can lead to severe consequences: based on it, an attacker can send a roundQC message with a nil QC field, causing any node on the entire chain to crash, or even forcing all nodes on the chain to go offline, thereby compromising the security and availability of the chain. This case was only found by Thor, which mutates in-flight roundQC requests using runtime context via the passive mode and changes the field QC to nil. Other tools lack the passive sending mode and can only mutate historical QC requests, and are thus rejected by early validations, and miss such bug.

6.6 Overhead of THOR

In this section, we evaluated the time overhead brought by Thor under three systems. We calculated the TPS of the system under the default workload provided by the corresponding systems. The results are listed in Table 6

Table 6. Runtime overhead of Thor across different blockchain systems. Throughput is measured in transactions per second (TPS) under default workloads.

System	Baseline TPS	TPS with THOR	Overhead (%)
FISCO BCOS	12,821	11,081	15.7%
Chainmaker	14,029	12,082	16.1%
Go-Ethereum	3,855	3,507	9.9%
Average	–	–	**13.9%**

Specifically, Thor introduces a 9.9%–16.1% drop in TPS across these systems, primarily due to the additional packet load imposed on consensus nodes. We consider this overhead acceptable, as it remains within the operational tolerance of enterprise deployments. The overhead is imported only in the testing environment where Thor is enabled, which will not be included in real-world production. As a testing and auditing framework, Thor does not introduce any additional logic into the production deployment of blockchain systems. In real-world use, the testing logic and oracle-related instrumentation are not included in the production chain, and therefore Thor incurs no runtime overhead on normal transaction execution or consensus in production. For the overhead to the CI/CD pipelines, we calculated the time used in FISCO BCOS. The total runtime of all CI/CD tasks recently ranges from 215–395 minutes. With Thor, this increases slightly by 14.8%–22.3%, still within CI budgets.

7 Lessons Learned

Through analyzing the 87 bugs uncovered by Thor across 9 enterprise-grade blockchain systems, we distilled several key lessons that can guide both developers and researchers in improving the reliability of blockchain infrastructure. These lessons span best practices for secure system development, strategies for effective CI/CD fuzzing integration, and the long-term maintenance of fuzzing results. We summarize three major takeaways below.

Lesson 1: Many critical bugs can be prevented through simple boundary checks. Our analysis shows that 63.2% of the vulnerabilities discovered by Thor could have been avoided with basic defensive programming practices. In particular, such bugs originate from missing boundary validations, such as unchecked message lengths, array index bounds, or integer overflows. Although these issues may seem minor, they can lead to severe consequences in enterprise-grade blockchain systems, including node crashes, consensus divergence, and state inconsistencies.

For instance, CVE-2022-28936 [40] in FISCO BCOS was resolved by introducing a check on the block's header index [10]. Similarly, bug#30968 [46] in Go-Ethereum was addressed by adding a null pointer check for the variable `info`. These examples highlight the importance of enforcing strict input validation and boundary condition checks as a standard development practice, especially in low-level protocol and message processing code. By adopting such practices early in the development cycle, developers can significantly reduce the likelihood of vulnerabilities and enhance the overall robustness of blockchain systems.

Lesson 2: CI/CD Fuzzing should be orchestrated across the system components, not just code changes. Our experience shows that 86.2% of the bugs discovered by Thor were not introduced by the most recent code changes. Instead, many bugs were triggered in older components due to new input sequences or altered execution orders. This suggests that CI/CD fuzzing should not be narrowly scoped to recently modified files, but should instead be orchestrated to cover the broader execution landscape across versions.

For example, the bug #2211 we described in the Table 5 was triggered after introducing the *fast view change* mechanism. While the new mechanism enabled nodes to switch views more efficiently, it inadvertently exposed a liveness issue in the legacy PBFT engine's message handler. Specifically, when an attacker sent a view change message with a big 'view' value, the node adopted this extreme view number. The root cause lay in the handler logic, which failed to validate the `view` field. This case illustrates how new features can activate bugs buried in legacy components, emphasizing the need for fuzzing across component boundaries and version histories.

Lesson 3: Fuzzing is not a one-off task, but requires sustainable triage and regression tracking. Through the deployment of Thor in real-world blockchain projects, we observed that many bugs—once fixed—tend to reappear in later commits or under slightly altered execution paths. This highlights a critical insight: effective fuzzing is not merely about discovering issues, but also about managing and maintaining the results over time. Without proper triage, deduplication, and regression tracking, fuzzing efforts may become inefficient or even misleading.

For instance, a bug in FISCO BCOS [14] was discovered by Thor in both version 2 and version 3. We first found the issue in version 2.8.0, caused by improper validation of the length field in a P2P packet. Although developers fixed it with a boundary check, a later refactoring in version 3.0 inadvertently removed the fix, causing the same bug to reappear. This highlights the need for bug traceability and auto-generated regression tests in fuzzing frameworks.

8 Related Work

Continuous Fuzzing. Continuous fuzzing continuously tests a program by monitoring the codebase for changes and triggers fuzzing tests when new code is added or modified. For example, Google's OSS-Fuzz [49] continuously tests more than 600 open-source projects with fuzzing and has found tens of thousands of bugs. Similarly, Microsoft also provides the continuous fuzzing service OneFuzz [34] to proactively harden software prior to release. OneFuzz can be baked into the CI/CD process and continuously monitor the development pipeline. For academic research, Klooster et al. [24] focus on improving fuzzing for CI/CD by minimizing redundant fuzzing activities and prioritizing resource allocation. CIDFuzz [55] tackles the common challenge in CI of handling frequent code changes that may not be effectively tested by traditional approaches. AFLChurn [56] concludes that 77% of the bugs reported by OSSFuzz are due to new code modifications. Thus, it conducts greybox fuzzing on newly changed code. WingFuzz [27] analyzes the obstacles of continuous fuzzing on DBMS, and perform constant testings.

Blockchain Fuzzing. Blockchain fuzzing can be categorized into three levels: the application level, the virtual machine level, and the system level. For the application level, many works focus on conduct fuzzing on smart contracts [19, 22, 32, 33, 44, 45, 50]. Besides, some works use the fuzzing technique to test a project. For example, Icy-Checker [54] uses fuzzing to identify state inconsistency (SI) bugs in Dapps. For the virtual machine level, tools like EVMFuzzer [18] generate transactions to test the implementation of Ethereum virtual machines. Fluffy [53] tests the VMs by performing multi-transaction fuzzing. Similarly, ByzzFuzz [51] focuses on randomized testing for BFT algorithms at the component level, primarily evaluating protocol logic in component-level. Meanwhile, some frameworks like EVM* [30, 31] propose reinforcement blockchain virtual machines to prevent dangerous behaviours during the runtime of blockchain transactions. For the system-level fuzzing, LOKI [28] and Tyr [7] perform state-aware fuzzing for blockchain consensus protocols. While Phoenix [29] conducts context-sensitive chaos testing on blockchain systems to detect resilience issues in complex enviroments of a blockchain node.

Main Difference. Compared to existing continuous fuzzing research, Thor is specifically tailored for enterprise-grade blockchain systems, which have complex behaviors and consensus protocols that differ from traditional software. Unlike OSS-Fuzz and OneFuzz that primarily focus on application-level or API fuzzing, Thor addresses the challenges unique to blockchain fuzzing. Furthermore, compared to prior blockchain fuzzers, Thor supports continuous integration scenarios by utilizing the state model in the blockchain's node, accelerating convergence by two-tiered parallel fuzzing and automating oracle adoption using LLMs.

9 Conclusion

This paper presents our experience in bridging the gap between academic fuzzing research and the practical needs of enterprise-grade blockchain systems. We identify three key obstacles to deployment: the difficulty of constructing effective state models, the slow convergence of traditional fuzzing within CI/CD workflows, and the complexity of adapting logical bug oracles across diverse blockchain platforms. To address these challenges, we propose Thor, a continuous fuzzing framework that integrates hybrid-mode message generation, two-tiered parallel fuzzing, and LLM-based oracle adaptation. Thor has been deployed on 9 commercial blockchain systems, uncovering 87 previously unknown vulnerabilities.

Acknowledgement

We thank the shepherd and reviewers for their valuable comments. This research is sponsored in part by the NSFC Program (No. 62502254, 62525207, U2441238) and China Postdoctoral Science Foundation (No. 2024M761690, GZC20240828).

References

[1] Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, Zekun Li, Avery Ching, and Dahlia Malkhi. 2022. Twins: BFT Systems Made Robust. In *25th International Conference on Principles of Distributed Systems (OPODIS '21)*, Vol. 217. Dagstuhl, Germany, 7:1–7:29. doi:10.4230/LIPIcs.OPODIS.2021.7

[2] BNB Chain. 2022. BNB Smart Chain. https://www.bnbchain.org/en/bnb-smart-chain. Official website. Accessed at March 7, 2026.

[3] Dan Boneh, Saba Eskandarian, Lucjan Hanzlik, and Nicola Greco. 2020. Single Secret Leader Election. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies* (New York, NY, USA) *(AFT '20)*. Association for Computing Machinery, New York, NY, USA, 12–24. doi:10.1145/3419614.3423258

[4] ChainFuzz. 2024. panic: runtime error: nil pointer in procRoundQC. https://git.chainmaker.org.cn/chainmaker/issue/-/issues/1170. ChainMaker GitLab issue #1170. Accessed at March 7, 2026.

[5] ChainMaker. 2025. Bugs triggered by replacing old nodes in the blockchain (startup and consensus). https://git.chainmaker.org.cn/chainmaker/issue/-/issues/1301. ChainMaker GitLab issue #1301. Accessed at March 7, 2026.

[6] ChainMaker. 2026. ChainMaker GitLab. https://git.chainmaker.org.cn/. Official GitLab instance. Accessed at March 7, 2026.

[7] Yuanliang Chen, Fuchen Ma, Yuanhang Zhou, Yu Jiang, Ting Chen, and Jia-Guang Sun. 2023. Tyr: Finding Consensus Failure Bugs in Blockchain System with Behaviour Divergent Model. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2517–2532. doi:10.1109/SP46215.2023.10179386

[8] Usman W. Chohan. 2021. The Double Spending Problem and Cryptocurrencies. *SSRN Electronic Journal* (2021). doi:10.2139/ssrn.3090174

[9] Consensys. 2026. GoQuorum: A permissioned implementation of Ethereum supporting data privacy. https://github.com/Consensys/quorum. Accessed at March 7, 2026.

[10] cyjseagull. 2022. check the block index. https://github.com/FISCO-BCOS/FISCO-BCOS/pull/2306. FISCO BCOS github pull request #2306. Accessed at March 7, 2026.

[11] cyjseagull. 2022. fix viewchange overflow. https://github.com/FISCO-BCOS/FISCO-BCOS/pull/2311. FISCO BCOS github pull request #2311. Accessed at March 7, 2026.

[12] Diem. 2021. Welcome to the Diem project. https://www.diem.com/en-us/. Homepage of the Diem project. Accessed at March 7, 2026.

[13] Ethereum. 2026. Ethereum.org: The complete guide to Ethereum. https://ethereum.org/en/. Ethereum project homepage. Accessed at March 7, 2026.

[14] fCorleone. 2021. The node may have a bug when dealing with unformatted packet and lead to a crash. https://github.com/FISCO-BCOS/FISCO-BCOS/issues/1951. GitHub issue #1951. Accessed at March 7, 2026.

[15] fCorleone. 2021. The nodes change view frequently and stop generating blocks. https://github.com/FISCO-BCOS/FISCO-BCOS/issues/2211. GitHub issue #2211. Accessed at March 7, 2026.

[16] fCorleone. 2022. Node crash while performing DMC Transfer testing. SIGSEGV signal occurred. https://github.com/FISCO-BCOS/FISCO-BCOS/issues/3271. GitHub issue #3271. Accessed at March 7, 2026.

[17] FISCO-BCOS. 2026. Commits · FISCO-BCOS/FISCO-BCOS. https://github.com/FISCO-BCOS/FISCO-BCOS/commits/release-3.16.1/. GitHub commits page for the release-3.16.1 branch. Accessed at March 7, 2026.

[18] Ying Fu, Meng Ren, Fuchen Ma, Heyuan Shi, Xin Yang, Yu Jiang, Huizhong Li, and Xiang Shi. 2019. EVMFuzzer: Detect EVM Vulnerabilities via Fuzz Testing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 1110–1114. doi:10.1145/3338906.3341175

[19] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 531–548. doi:10.1145/3319535.3363230

[20] Hyperledger Caliper. 2026. A blockchain benchmark framework to measure performance of multiple blockchain solutions. https://github.com/hyperledger-caliper/caliper. GitHub repository for a blockchain benchmark framework. Accessed at March 7, 2026.

[21] Hyperledger Fabric. 2026. A Blockchain Platform for the Enterprise. https://hyperledger-fabric.readthedocs.io/en/release-2.5/. Project documentation for an enterprise-grade permissioned distributed ledger platform. Accessed at March 7, 2026.

[22] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. Association for Computing Machinery, New York, NY, USA, 259–269. doi:10.1145/3238147.3238177

[23] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, ON, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2123–2138. doi:10.1145/3243734.3243804

[24] Thijs Klooster, Fatih Turkmen, Gerben Broenink, Ruben Ten Hove, and Marcel Böhme. 2023. Continuous Fuzzing: A Study of the Effectiveness and Scalability of Fuzzing in CI/CD Pipelines. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*. IEEE, 25–32. doi:10.1109/SBFT59156.2023.00015

[25] L. Lamport. 1977. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering* SE-3, 2 (1977), 125–143. doi:10.1109/TSE.1977.229904

[26] Huizhong Li, Yujie Chen, Xiang Shi, Xingqiang Bai, Nan Mo, Wenlin Li, Rui Guo, Zhang Wang, and Yi Sun. 2023. FISCO-BCOS: An Enterprise-Grade Permissioned Blockchain System with High-Performance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23)*. Association for Computing Machinery, New York, NY, USA, 1–17. doi:10.1145/3581784.3607053

[27] Jie Liang, Zhiyong Wu, Jingzhou Fu, Yiyuan Bai, Qiang Zhang, and Yu Jiang. 2024. WINGFUZZ: Implementing Continuous Fuzzing for DBMSs. In *Proceedings of the 2024 USENIX Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC '24)*. USENIX Association, USA, 479–492. doi:10.5555/3691992.3692022

[28] Fuchen Ma, Yuanliang Chen, Meng Ren, Yuanhang Zhou, Yu Jiang, Ting Chen, Huizhong Li, and Jiaguang Sun. 2023. LOKI: State-Aware Fuzzing Framework for the Implementation of Blockchain Consensus Protocols. In *Proceedings of the Network and Distributed System Security Symposium (NDSS '23)*. The Internet Society. doi:10.14722/ndss.2023.24078

[29] Fuchen Ma, Yuanliang Chen, Yuanhang Zhou, Jingxuan Sun, Zhuo Su, Yu Jiang, Jiaguang Sun, and Huizhong Li. 2023. Phoenix: Detect and Locate Resilience Issues in Blockchain via Context-Sensitive Chaos. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 1182–1196. doi:10.1145/3576915.3623071

[30] Fuchen Ma, Ying Fu, Meng Ren, Mingzhe Wang, Yu Jiang, Kaixiang Zhang, Huizhong Li, and Xiang Shi. 2019. EVM: From Offline Detection to Online Reinforcement for Ethereum Virtual Machine. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 554–558. doi:10.1109/SANER.2019.8668038

[31] Fuchen Ma, Meng Ren, Ying Fu, Mingzhe Wang, Huizhong Li, Houbing Song, and Yu Jiang. 2021. Security reinforcement for Ethereum virtual machine. *Inf. Process. Manage.* 58, 4 (July 2021), 14 pages. doi:10.1016/j.ipm.2021.102565

[32] Fuchen Ma, Zhenyang Xu, Meng Ren, Zijing Yin, Yuanliang Chen, Lei Qiao, Bin Gu, Huizhong Li, Yu Jiang, and Jiaguang Sun. 2022. Pluto: Exposing Vulnerabilities in Inter-Contract Scenarios . *IEEE Transactions on Software Engineering* 48, 11 (Nov. 2022), 4380–4396. doi:10.1109/TSE.2021.3117966

[33] Fuchen Ma, Zhenyang Xu, Meng Ren, Zijing Yin, Yuanliang Chen, Lei Qiao, Bin Gu, Huizhong Li, Yu Jiang, and Jiaguang Sun. 2022. Pluto: Exposing Vulnerabilities in Inter-Contract Scenarios . *IEEE Transactions on Software Engineering* 48, 11 (Nov. 2022), 4380–4396. doi:10.1109/TSE.2021.3117966

[34] Microsoft. 2024. OneFuzz: A self-hosted Fuzzing-As-A-Service platform. https://github.com/microsoft/onefuzz. GitHub repository for the OneFuzz platform. Accessed at March 7, 2026.

[35] MITRE. 2022. CVE-2021-46359. https://www.cve.org/CVERecord?id=CVE-2021-46359. FISCO-BCOS denial-of-service vulnerability that may enable double-spending attacks. Accessed at March 7, 2026.

[36] MITRE. 2022. CVE-2022-26298. https://www.cve.org/CVERecord?id=CVE-2022-26298. Reserved by a CVE Numbering Authority (CNA). Public details not yet available. Accessed at March 7, 2026.

[37] MITRE. 2022. CVE-2022-26300. https://www.cve.org/CVERecord?id=CVE-2022-26300. EOS v2.1.0 was discovered to contain a heap-buffer-overflow via the function txn_test_gen_plugin. Accessed at March 7, 2026.

[38] MITRE. 2022. CVE-2022-26534. https://www.cve.org/CVERecord?id=CVE-2022-26534. FISCO-BCOS release-3.0.0-rc2 vulnerability where a malicious node can use a malicious viewchange packet to crash normal nodes. Accessed at March 7, 2026.

[39] MITRE. 2022. CVE-2022-28936. https://www.cve.org/CVERecord?id=CVE-2022-28936. FISCO-BCOS release-3.0.0-rc2 was discovered to contain an issue where a malicious node can trigger an integer overflow and cause a denial of service. Accessed at March 7, 2026.

[40] MITRE. 2022. CVE-2022-28936. https://www.cve.org/CVERecord?id=CVE-2022-28936. FISCO-BCOS release-3.0.0-rc2 was discovered to contain an issue where a malicious node can trigger an integer overflow and cause a Denial of Service (DoS) via an unusually large viewchange message packet. Accessed at March 7, 2026.

[41] MITRE. 2022. CVE-2022-28937. https://www.cve.org/CVERecord?id=CVE-2022-28937. FISCO-BCOS release-3.0.0-rc2 vulnerability where a malicious node can use an invalid proposal with an invalid signature to cause normal nodes to crash. Accessed at March 7, 2026.

[42] MITRE. 2022. CVE-2022-45196. https://www.cve.org/CVERecord?id=CVE-2022-45196. Hyperledger Fabric 2.3 allows attackers to cause a denial of service (orderer crash) by repeatedly sending a crafted channel transaction. Accessed at March 7, 2026.

[43] ngotchac. 2024. Missing some block hash -> block number links in DB after restart | "block body not found" error. https://github.com/ethereum/go-ethereum/issues/30119. GitHub issue #30119. Accessed at March 7, 2026.

[44] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Minh Tran. 2020. sFuzz: An Efficient Adaptive Fuzzer for Solidity Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Association for Computing Machinery, New York, NY, USA, 778–788. doi:10.1145/3377811.3380334

[45] Meng Ren, Fuchen Ma, Zijing Yin, Huizhong Li, Ying Fu, Ting Chen, and Yu Jiang. 2021. SCStudio: a secure and efficient integrated development environment for smart contracts. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 666–669. doi:10.1145/3460319.3469078

[46] sdjasj. 2024. Continuously changing configurations and restarting the node caused the node to panic during runtime. https://github.com/ethereum/go-ethereum/issues/30968. GitHub issue #30968. Accessed

at March 7, 2026.

[47] SecTechTool. 2021. Nodes crash down after receiving a serial of messages generated by fuzzer, and cannot be recovered. https://github.com/ethereum/go-ethereum/issues/23866. GitHub issue #23866. Accessed at March 7, 2026.

[48] Sei. 2026. Sei | The Fastest EVM Blockchain for High-Frequency Apps. https://www.sei.io/. Homepage of the Sei project. Accessed at March 7, 2026.

[49] Kostya Serebryany. 2017. OSS-Fuzz – Google's continuous fuzzing service for open source software. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany. Presentation at the 26th USENIX Security Symposium, Vancouver, BC, Canada. Accessed at March 7, 2026.

[50] Chaofan Shou, Shangyin Tan, and Koushik Sen. 2023. ItyFuzz: Snapshot-Based Fuzzer for Smart Contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA '23)*. Association for Computing Machinery, New York, NY, USA, 322–333. doi:10.1145/3597926.3598059

[51] Levin N. Winter, Florena Buse, Daan de Graaf, Klaus von Gleissenthall, and Burcu Kulahcioglu Ozkan. 2023. Randomized Testing of Byzantine Fault Tolerant Algorithms. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 101 (April 2023), 32 pages. doi:10.1145/3586053

[52] Brent Xu, Dhruv Luthra, Zak Cole, and Nate Blakely. 2018. EOS: An Architectural, Performance, and Economic Analysis. https://cdn0.tnwcdn.com/wp-content/blogs.dir/1/files/2018/11/EOS_Report.pdf. Whiteblock technical report. Accessed at March 7, 2026.

[53] Youngseok Yang, Taesoo Kim, and Byung-Gon Chun. 2021. Finding Consensus Bugs in Ethereum via Multi-transaction Differential Fuzzing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 349–365. https://www.usenix.org/conference/osdi21/presentation/yang

[54] Mingxi Ye, Yuhong Nan, Zibin Zheng, Dongpeng Wu, and Huizhong Li. 2023. Detecting State Inconsistency Bugs in DApps via On-Chain Transaction Replay and Fuzzing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA '23)*. Association for Computing Machinery, New York, NY, USA, 298–309. doi:10.1145/3597926.3598057

[55] Jiaming Zhang, Zhanqi Cui, Xiang Chen, Huiwen Yang, Liwei Zheng, and Jianbin Liu. 2023. CIDFuzz: Fuzz Testing for Continuous Integration. *IET Software* 17, 3 (June 2023), 301–315. doi:10.1049/sfw2.12125

[56] Xiaogang Zhu and Marcel Böhme. 2021. Regression Greybox Fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2169–2182. doi:10.1145/3460120.3484596