Enhancing Blockchain Robustness through Efficient Feedback-Driven Chaos Engineering

Junjie Ma[†§‡], Muhui Jiang[¶], Edmond W.W.Chan[‖], Xiapu Luo[‡*], Qi Wang[§†*], Fengwei Zhang[§†*]
[†]Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China
[§]Department of Computer Science and Engineering, Southern University of Science and Technology, China
[‡]Department of Computing, The Hong Kong Polytechnic University, China
[¶]BlockSec, China [‖]Akamai Technologies, USA

***Abstract*—The rapid evolution of blockchain technology has attracted global users to operate nodes and participate in consensus processes. However, heterogeneous networks and hardware conditions may induce chaotic behavior in nodes, leading to critical faults such as packet loss or data corruption. These faults compromise system robustness, causing node crashes or desynchronization. Existing chaos engineering methods for blockchain systems suffer from inefficient fault injection and lack feedback mechanisms to iteratively guide fault selection.**

To address these limitations, we propose ChaosChain, a framework for efficient and effective chaos engineering in blockchain systems to expose robustness vulnerabilities. For efficiency, we design a parallel fault coordination mechanism that injects multiple fault types into arbitrary nodes simultaneously while enforcing blockchain consensus security constraints. We construct a global timeline of all executed faults and validate security constraints before each injection. To improve effectiveness, we introduce next fault guidance, a feedback-driven approach that selects optimal faults to trigger complex node state transitions. We model each node's state as an interference-resistant state machine that captures state transitions during faults without cross-node interference.

We evaluate ChaosChain across four implementations in Filecoin and Ethereum, uncovering six new robustness issues. Compared to state-of-the-art tools, our framework injects 4× more faults concurrently.

I. Introduction

With the rapid growth of the blockchain ecosystem, many well-known public blockchains have emerged, including Ethereum [7], Filecoin [12], and Polkadot [53]. These blockchains allow users worldwide to set up their nodes. These nodes can join the blockchain network and participate in the consensus process. However, this inclusivity can introduce chaos into the blockchain system. Each node operates on its own local network and hardware environment, leading to potential faults such as network fluctuations that cause packet loss when exchanging messages with nodes or hardware failures that prevent nodes from accessing parts of their stored data. Thus, the blockchain system employs consensus mechanisms [33] to tolerate faulty nodes and ensure that all nodes maintain the same state.

* Xiapu Luo, Qi Wang, and Fengwei Zhang are the corresponding authors.

However, considering the complex real-world scenarios, maintaining such robustness can be challenging for the blockchain system [78]. For example, Ethereum has over 6,500 nodes from more than 50 countries [10]. In 2023, it faced a robustness issue when some nodes continued to broadcast outdated attestations to the network, possibly due to network delays [54]. This caused other nodes to suffer a DoS attack, resulting in a 30-minute halt in the generation of new blocks [48]. This incident resulted in a total loss of over $200,000.

Chaos engineering is considered a novel methodology for assessing software system robustness [1]. This approach involves deliberately introducing faults into the system to evaluate whether it can recover to its steady state. Chaos engineering has proven to be a useful tool within blockchain systems. For example, *CHAOSETH* [76] evaluates the Ethereum implementation's resilience under chaotic system call results. *Phoenix* [38] detects blockchain resilience issues under network chaos. *Attacknet* [25] injects different faults into the Ethereum system to identify possible failures. However, these methods often fall short of performing efficient and effective chaos engineering for several reasons.

First, inefficiency in fault injections towards the blockchain system. These existing methods inject only one type of fault into nodes in the blockchain system at a time. This makes the testing process time-consuming and cannot simulate real-world, complex scenarios in which multiple nodes may experience different faults simultaneously. Although general-purpose chaos engineering tools like Gremlin [20] and Litmus [35] support multiple fault types, they lack blockchain-specific capabilities, such as awareness of the consensus state.

Second, the selection range of nodes for fault injection in these methods is limited. *CHAOSETH* only selects one node for fault injection, while *Phoenix* and *Attacknet* require manual effort to pre-define which fixed nodes can be injected with faults, rather than allowing for flexible selection of any nodes within the blockchain system. This limitation makes it difficult to perform efficient chaos engineering to identify robustness issues hidden deep within the system.

Third, these methods lack an effective feedback mechanism to guide fault injection. They select faults to inject into the system manually or based on pre-defined states rather than being guided by the results of previous faults. This can lead to inefficient testing to uncover robustness issues.

However, addressing these issues presents two challenges.

The first challenge is maintaining the consensus steady state while injecting multiple fault types simultaneously across an arbitrary number of nodes. To effectively simulate real-world blockchain fault scenarios in chaos engineering, we must inject multiple faults into multiple nodes simultaneously. However, each blockchain system has security constraints inherent to its consensus protocol. For example, the consensus protocol used by Ethereum requires that no more than 49% of the miner nodes exhibit misbehavior [8]. If more than half of the miner nodes are injected with faults, it would generate an uncorrectable hard fork, and the system would not be able to return to its steady state. Such a steady state violation is meaningless due to a breach of the consensus protocol. Therefore, a fault-injection strategy must consider these security constraints before effectively injecting multiple faults across multiple nodes.

The second challenge is designing an effective feedback mechanism to select suitable faults that trigger deeply hidden robustness issues. A feedback mechanism is needed to effectively extract the node's state transitions during the fault injection period and monitor the fault's impact on the node's state. Although previous methods propose metrics such as node state measurements based on built-in monitoring metrics [76] or node consensus states [38], these metrics are ineffective under multiple fault scenarios. The metrics from built-in monitoring components do not account for the impact of other nodes, making them unsuitable for scenarios with multiple nodes under fault injection. On the other hand, the node consensus state is not fine-grained enough to monitor node state changes to guide feedback for selecting new fault injections. For example, if an injected fault causes network message traffic to increase significantly, this metric cannot capture the state changes since the node's consensus state remains the same. Thus, the feedback mechanism must capture state transitions precisely and be fine-grained enough to identify state transitions during the fault period.

To address the aforementioned challenges, we propose ChaosChain, a framework that identifies robustness issues in blockchain systems by introducing parallel fault coordination and next fault guidance via reinforcement learning. To address the first challenge, we construct a global timeline of all executing faults and apply security constraints to ensure that injected faults do not violate the requirements of the corresponding blockchain consensus protocol. For the second challenge, we model the node state as a Mealy state model to capture state transitions during testing. This model serves as a metric to guide the reinforcement learning method in selecting the next suitable fault to inject into the blockchain system. Regarding the fault types, we not only inject crash faults [68], such as network chaos, but also introduce blockchain-related Byzantine faults [3], such as double-spending [71], to simulate complex node behavior within the blockchain system. By implementing these approaches, our framework can achieve effective and efficient chaos engineering within the blockchain system.

Our contributions can be summarized as follows:

Efficient Fault Injection. We propose a novel approach that combines fault timelines with security constraints from the consensus protocol, allowing the injection of multiple types of faults simultaneously on an arbitrary number of nodes. The evaluation shows that this approach can inject three times as many faults into the blockchain system and four times as many faults simultaneously as existing methods.

Effective Node State Modeling. To provide an effective metric for the guidance to select suitable faults for each node to achieve maximum state diversity, we construct the node Mealy state machine using the timeline. This metric ensures that the state is not disrupted by other nodes under different faults. The evaluation shows that our state machine provides consistent and effective results under complex fault scenarios.

Comprehensive Evaluation and Implementation of ChaosChain. We implement and evaluate the effectiveness of our framework on Filecoin and Ethereum. ChaosChain detected six new robustness issues, four of which have been confirmed by developers. We will open-source the framework for further examination and validation.

II. Background

A. Chaos Engineering

Chaos engineering is a software testing methodology aimed at evaluating a software system's ability to tolerate failures while still providing service [70], [1]. Netflix first popularized this method with a tool called *ChaosMonkey*, which randomly shuts down servers related to production services to assess whether the system continues to function properly [43].

Chaos engineering is designed based on four principles [1]: First, build a hypothesis around the system's steady state. To evaluate the effectiveness of the injected chaos in disrupting the system's normal service, we need to formally define the system's normal state as the steady state. For example, in a blockchain system, we can define a node's steady state as its ability to sync with the latest block and produce new blocks when available. After defining the system's steady state, we need to determine the types of chaos to be injected into the system to simulate real-world failure events. For example, we can introduce node-offline scenarios or noisy network traffic as potential faults to inject chaos. Third, we need to inject chaos into the production environment, which requires reproducing the entire system in a test context. However, in blockchain systems, replicating the main chain is impractical. Instead, we can replicate the local blockchain with the same configuration as the main chain to simulate the production environment. Finally, it is essential to automate experiments to run continuously. Automation in chaos engineering is crucial because the system is continuously evolving. Continuous testing ensures that experiments run repeatedly as the system evolves and that new production services are designed to withstand these failures. This ongoing process helps maintain system resilience over time.

In blockchain systems, to better evaluate robustness, we should incorporate Byzantine faults alongside crash faults, allowing nodes to misbehave. This approach enables us to simulate the complex node behaviors within blockchain systems and provide a more comprehensive assessment of their robustness.

B. Blockchain System Robustness

The blockchain system is maintained by individual nodes that adhere to the same consensus protocol [8]. Compared to consensus protocols used within distributed database systems like Paxos [31] or Raft [49], which only consider crash fault tolerance (CFT) where nodes can crash but the system remains operational [68], blockchain consensus protocols allow for Byzantine Fault Tolerance (BFT) [3]. The BFT can handle nodes performing malicious behavior, such as broadcasting invalid transactions. Additionally, the concept of client diversity [79], proposed by Ethereum, requires nodes within the blockchain system to have multiple implementations across different programming languages. For example, Prysm [56] is implemented in Golang [18], while Lighthouse [61] is implemented in Rust [58]. This diversity makes the real-world blockchain system a complex peer-to-peer network that must be robust enough to tolerate not only multiple types of faults but also a variety of node implementations.

However, once the system's robustness is violated, it can cause significant damage. For example, the Ethereum consensus network experienced a robustness bug on May 11, 2023 [48]. The blockchain failed to finalize due to missing blocks. Approximately 149 blocks were missing, resulting in a total loss of 55 ETH. The issue was caused by nodes failing to handle out-of-date attestations, leading the affected nodes to continuously rebroadcast these outdated messages. This caused other normal nodes to run out of resources due to the expensive operations required to process them.

Thus, chaos engineering for blockchain systems must not only cover malicious node behavior but also be flexible enough to account for nodes with different implementations, simulating the complex behavior of real-world blockchain systems.

III. Overview

```
1  func (ms *msgSet) add(m *SignedMessage, ...) {
2      exms := ms.msgs[m.Message.Nonce]
3      if m.Cid() == exms.Cid() {
4          return ErrExistingNonce
5      }
6  }
7  func (mv *MessageValidator) Validate() {
8      err := mv.mpool.Add(ctx, m)
9      switch err {
10     case ErrExistingNonce:
11         return ValidationReject
12     }
13 }
14 func (p *PubSub) handleIncomingRPC(rpc *RPC) {
15     switch p.rt.AcceptFrom(rpc.from) {
16     case AcceptNone:
17         log.Debugf("dropping RPC", rpc.from)
18         return
19     }
20 }
21 func (gs *GossipSubRouter) AcceptFrom(p peer.ID) {
22     if Score(p) < graylistThreshold {
23         return AcceptNone
24     }
25 }
```

Fig. 1: Simplified code snippet demonstrating the sync failure bug in Filecoin.

A. Threat Model

We define the blockchain system B under chaos engineering as the tuple (N, C, R, T, F). The N represents the number of blockchain nodes in the network. And $C \in N$ represents the nodes currently injected with chaos faults. The $R \in N$, where $C \cap R = \emptyset$, means the set of nodes that have stopped being injected with chaos faults and are under recovery back to a steady state. The symbol T represents the transactions of native tokens between nodes. The F represents the chaos faults that can be injected into the nodes within $\{N - R\}$ in the blockchain network. These faults include crash faults, such as network interruptions, and Byzantine faults, such as double-spending transactions, where $tx, tx' \in TX$ have the same nonce.

We define blockchain robustness issues as disruptions to the steady state of any node within the blockchain system. We formally define two types of node steady states: **functionality steady state** and **consensus steady state**. The **functionality steady state** of node n_i is defined as follows.

$$\forall n_j \in \{N - C - R\}, n_i.blocks = n_j.blocks \tag{1}$$

This ensures that the node n_i can synchronize with the latest blocks from most normal nodes. If the node holds a block number different from the majority, its functionality steady state is considered disrupted. The **consensus steady state** of node n_i is defined below.

$$\begin{aligned} \forall n_j \in \{N - C - R\}, n_i.balance(n_i) = n_j.balance(n_i) \\ \wedge n_i.balance(n_j) = n_j.balance(n_j) \end{aligned} \tag{2}$$

This ensures that there is no hard fork within node n_i, as it holds the same consensus results as the majority of normal nodes. If n_i holds a different value, its consensus steady state is considered disrupted. If either the functionality steady state or the consensus steady state is disrupted, we consider the node to have a robustness issue.

We define our chaos engineering strategy S on top of the blockchain network B as the tuple $(B, FG, FC, SC, l) \implies ((N, C, R, T, F), FG, FC, SC, \tau)$. The $FG : N \times F \to F'$ represents the fault guidance function. It selects nodes based on the security assumption constraints to inject suitable faults. The $FC : N \times F \times SC \to C$ is the fault coordination function, it injects the faults selected by the fault guidance function into the blockchain network. The execution sequence is coordinated by the corresponding consensus protocol's security constraints SC, ensuring that injected faults do not deliberately corrupt the node's steady state by causing illegal executions, such as a 51% attack [63] or an eclipse attack [23], [65]. Our chaos engineering strategy aims to select suitable faults to inject into the system, causing nodes to be unable to recover to their steady state within the recovery time limit τ.

B. Motivation Example

In this section, we discuss a known robustness bug in the Filecoin blockchain system that causes nodes to fail to sync the latest blocks from the network [64].

The root cause of the issue is the incorrect penalization of peers who resend duplicate messages. As illustrated in

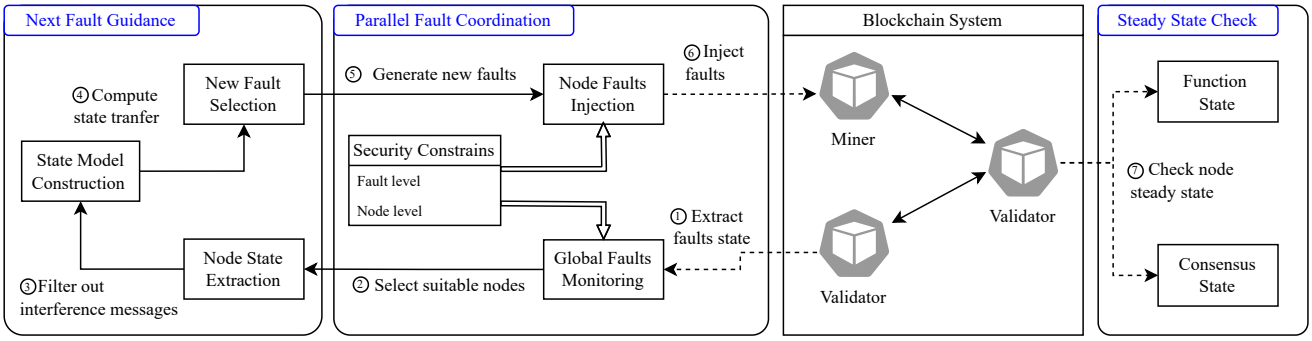


Fig. 2: The overview of the ChaosChain framework. The **parallel fault coordination** is responsible for capturing and modeling all the injected fault states within the blockchain system. The **next fault guidance** captures feedback from the nodes within the blockchain system. The **steady state check** verifies whether a node fails to recover to its steady state.

the Figure 1, in line 1, the function *add* processes received messages from other nodes. If a remote node *validates* that the received message has the same nonce as an already received message (line 9), it will generate an *ErrExistingNonce* error, reject the message, and decrease the sender node's score below the *graylistThreshold*. This causes the remote node to reject any future messages from the sender node.

Thus, if the sender node has pending messages that persist for a long time, it will republish them at regular intervals, leading to penalties from all its neighboring nodes. However, the sender node can still receive the latest blocks via random broadcasts from its neighbors. Once it misses a block due to network chaos, it cannot request the missing block from any neighbor, since they reject its requests. Thus, the sender node might not be synced with the latest chain due to a single missing block.

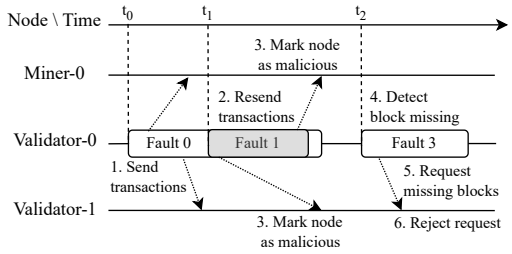


Fig. 3: The timeline of the Lotus sync failure bug. The grey box indicates that both faults are executed simultaneously.

The Figure 3 illustrates the timeline for reproducing such a bug. The node *validator-0* is our victim node. The blockchain network contains three nodes: miner-0, validator-0, and validator-1. Miner-0 is the block producer, while validator-0 and validator-1 are normal nodes without mining power.

At time t_0, validator-0 starts sending two transactions with the same nonce due to the injected fault of double-spending. After time t_1, the system time is altered by the injected fault, leading it to believe its transaction was dropped in the network and prompting it to resend the transactions.

After receiving the duplicate transactions, validator-1 and miner-0 perceive validator-0 as malicious and reject any requests from it. Despite this, validator-0 can still sync with the latest blocks since miner-0 and validator-1 can still broadcast new blocks to validator-0.

Next, after t_2, validator-0 misses one of the new blocks from either miner-0 or validator-1 due to the injected network chaos. Consequently, validator-0 starts requesting the missing block from validator-1 and miner-0. Since they reject validator-0's request, validator-0 falls out of sync due to the missing block and gets stuck at syncing, even though it can still receive the latest blocks.

IV. Approach

The overview of our framework ChaosChain is shown in Figure 2. The framework consists of three main components: textbf parallel fault coordination, **next fault guidance**, and **steady state check**. The whole chaos engineering process starts as follows. After setting up the blockchain system, we first ① check the fault state within the blockchain system and construct the global fault timeline within the global fault monitoring. We then ② select all possible nodes that can be injected with new faults, based on the node's security constraints, to obtain the node state information. Then, in ③, we filter out interference messages from other nodes to construct the node state model. Next, in ④, we compute the node state transitions to select suitable faults for the new fault selection. In ⑤, the selected new faults are sent to the node fault injection module to wait for the appropriate time to be injected into the system. Then, in ⑥, the faults are injected into the system properly under the constraints. Finally, at ⑦, after each fault ends, our steady state check verifies whether the node has recovered to its steady state.

A. Parallel Fault Coordination

In the parallel fault coordination component, we first monitor all fault events within the blockchain system by constructing a timeline of all executed faults in causal order. Then, in the node fault injection module, we apply a constraint check with the timeline to determine the appropriate timing for fault injection. This ensures that the executed fault would not violate the node security requirements.

1) Global Faults Monition: We first build a global, real-time timeline of the blockchain state to capture which nodes are experiencing faults and the corresponding fault types. Additionally, such a timeline must be prefix-closed [39] to capture the causal order of both executing and executed faults. This means that if one fault starts execution, we must extract all the finished faults and all the currently executing faults.

However, constructing such a real-time, prefix-closed causal order timeline within the blockchain system for chaos engineering involves two issues: a) Indeterminate ending time of injected faults. Although we can predetermine the fault execution time, the recovery time is difficult to predict. A node might take several minutes to recover to a steady state because it fetches too many missing blocks. Consequently, we cannot use dependency graphs to construct the timeline without a determined test case duration. b) Inconsistent node time clocks. During the test period, we might inject faults such as system time skew, which can result in incorrect system times. Therefore, we cannot obtain a synchronized real-time clock across different nodes.

To address these issues, we construct the timeline based on the fault execution durations. As shown in Algorithm1, to obtain the edges of fault execution duration (starting and ending times), we capture signals from the nodes at the fault's beginning and end by instrumenting two functions. When a node starts executing a fault, a start event is sent to our logger. After the node recovers from the fault, an end event is sent back to the logger. Thus, this method eliminates the need to continuously query the node's fault execution state to determine whether the fault has finished.

To overcome the lack of synchronized real-time clocks, we use the Lamport timeline [32] to capture the order of events. The Lamport timeline adopts logical clocks to maintain a consistent order of events. Before testing, the coordinator holds an incrementing counter. Whenever the coordinator injects a fault into a node, it increments its logical clock. When the node executes the fault, it sends back a start event with the incremented clock value. Upon the fault ending, it sends an end event with another incremented clock value. Upon receiving an event from a node, the logger sets its clock to the greater of its current clock and the event's clock value. After that, the clock is incremented before processing any further events. The Lamport timeline eliminates the issue of unsynchronized real-time clocks. Compared to other logical timeline methods, such as vector clocks [62] or logical clocks [57], the Lamport timeline offers simplicity and low overhead. We only need to modify each node to send events rather than maintain a vector of counters for all nodes.

2) Node Fault Injection: After capturing the real-time fault states, we can inject faults into the blockchain system. To simulate real-world complex node behaviors and expedite the chaos engineering process, we allow multiple types of faults across different nodes simultaneously, as well as multiple types of faults on a single node. This approach enables us to explore deep, complex states within the blockchain system and individual nodes.

Unlike other consensus protocols used in distributed databases, such as Paxos [31] and Raft [49], which are designed only to handle crash faults where a node halts execution prematurely [68], blockchain consensus protocols must handle more than just crash faults. For example, they must address issues such as double-spending [17]. Consequently, our framework should be able to inject Byzantine faults and crash faults, unlike those used only for distributed databases [41].

Algorithm 1: Construct the global fault timeline.

```
  /* timeline construction                    */
1 timeline.init();
2 clock = 0;
3 while nodeID, states, Rclock <- ReceiveEvent do
4 |   logicClock = MAX(Rclock, clock);
5 |   timeline.Log(nodeID, states, clock);
6 |   logicClock ++;
  /* node fault injection execution           */
7 Function BeforeTestcase():
8 |   logicClock ++ ;
9 |   SendEvent(nodeID, START, clock)
10 Function AfterTestcase():
11 |   logicClock ++;
12 |   SendEvent(nodeID, FINISH, clock)
```

However, the consensus protocols used within blockchain systems have their own security requirements [17]. For example, the Ethereum beacon chain uses the proof-of-stake [66] consensus protocol, which requires at least 51% of validators to be benign [63], similar to the proof-of-work [67] mechanism. Therefore, there exists a boundary for conducting chaos engineering within blockchain systems through fault injection. Exceeding this boundary by injecting excessive faults may compromise the system's ability to maintain consensus and overall security.

Unconstrained chaos engineering can introduce unrecoverable system failures. To illustrate this, we present a small test case on the Filecoin network comprising 3 miners and 9 validators. In typical chaos engineering, the goal is to inject as many faults as possible into the network. This could lead to scenarios in which all three miner nodes (A, B, and C) experience simultaneous faults, such as being offline, clock skew, or packet loss. Such a situation results in two separate hard forks in the blockchain.

- Miner A believes the network has terminated, since it appears offline to all other nodes.
- Miner B perceives the other two miners as corrupted (due to their failure to generate valid blocks at the required time) and assumes it is the only valid miner, thus continuing to broadcast new blocks.
- Miner C believes itself is the only valid miner (seeing A as offline and B as producing invalid blocks) and begins broadcasting blocks for future time slots.

When all faults are resolved, according to Filecoin's consensus rules, both chains produced by miners B and C are considered valid. Consequently, validators will follow either miner, depending on the block-broadcast sequence, resulting in a hard

fork that cannot be recovered from. Importantly, this fault scenario does not violate Filecoin's consensus protocol and therefore cannot be classified as a bug, since the majority of nodes were simultaneously corrupted.

Thus, it is crucial to consider these security constraints when designing fault injection scenarios. By ensuring that fault injections remain within the system's security boundaries, we can maintain the system's steady state and integrity. This careful balance allows us to effectively test the system's robustness without undermining its fundamental security guarantees.

Thus, designing rules that maintain the security constraints while performing chaos engineering presents several challenges: a) Flexibility in node selection: The nodes selected for injected faults must be flexible. We allow arbitrary fault injections across all nodes in the blockchain system rather than limiting them to predefined nodes. b) Parallel fault execution: We allow faults to be injected into multiple nodes simultaneously, and a single node might suffer multiple types of faults.

To address these issues, we convert the security requirements into corresponding constraints on the fault types and the number of faults that can be executed simultaneously in the blockchain system. Specifically, we apply two levels of restriction: node-level and fault-level. Along with the global fault timeline, these constraints check whether the selected nodes and faults are suitable for injection.

Within the node-level constraint, we aim to limit the number of nodes that can be injected with faults to ensure that the consensus protocol's security constraints are not violated. For example, the PoS consensus used in Ethereum requires at least 51 percent of the miner nodes to be benign [66]. Thus, we translate the constraint into the restriction shown in Constraint 1 (Equation 3), where N_{miner} is the set of all the miner nodes in the network, and $fault(n_i)$ is a boolean function representing whether a node has been injected with a fault.

$$\sum_{n_i \in N_{miner}} fault(n_i) < \left\lfloor \frac{N_{miner}}{2} \right\rfloor \tag{3}$$

This ensures that at most 49% of the miner nodes are faulty, thereby maintaining the security requirements of the consensus protocol.

As for the fault-level constraint, we limit the types and number of faults that can be injected into a node based on the node's consensus requirements. To prevent a node from suffering an eclipse attack [23], [65] due to injected faults, where all its network connections to remote peers become disrupted. We implement the following Constraint 2 (Equation 4) to ensure that each node has at least one connection to a benign remote peer. Here, N represents the set of all nodes, and $benign\_connection(n_i, n_j)$ is a boolean function indicating whether there is a benign connection between two nodes n_i and n_j.

$$\forall n_i \in N, \quad \sum_{\substack{n_j \in N \\ n_j \neq n_i}} \text{benign\_connection}(n_i, n_j) > 1 \tag{4}$$

To prevent injecting the same faults into the same node repeatedly, we apply the following Constraint 3 (Equation 5) to ensure that the same type of fault is not injected into the same node twice. Here, F is the set of all fault types, and $inject(n_i, f_j)$ is a boolean function indicating whether the fault f_j has been injected into node n_i.

$$\forall n_i \in N, \sum_{\forall f_j \in F} inject(n_i, f_j) \leq 1 \tag{5}$$

To prevent conflicting faults that might execute simultaneously and cause undesired effects, we apply the following Constraint 4 (Equation 6).

$$\text{if } inject(n_i, \text{"offline"}) = 1, \sum_{\forall f_j \in F} inject(n_i, f_j) = 1 \tag{6}$$

This constraint ensures that an "offline" fault can be injected only when the node is not already under any other faults. This prevents the coincidence of offline faults with other faults that could result in invalid testing.

Algorithm 2: Inject faults to the blockchain system.

```
Data: Faults
chainStatus = FetchChainStatus();
for fault ∈ Faults do
    while !CheckFault(fault, chainStatus) do
        Wait();
    InjectFault(fault);
    chainStatus = UpdateChainStatus();
Function CheckFault(fault, chainStatus):
    for constrain ∈ Constrains do
        if !constrain.Check(fault, chainStatus) then
            Return false;
    Return true;
```

As shown in Algorithm 2, we first assess the current fault state using the global Lamport timeline to inject new faults into the blockchain system. Then we check whether the new fault's injection violates any previously established constraints. We inject it into the corresponding nodes if it does not violate any restrictions. Otherwise, we will continue assessing the current fault states until the injection is valid.

B. Next Fault Guidance

In the next fault guidance components, based on the parallel fault coordination timeline, we select suitable nodes to inject new faults. First, we obtain the node network messages to construct the node state model. Then, we analyze the state transitions within the state model to evaluate the effectiveness of previously injected faults in triggering new states. Finally, we apply reinforcement learning to choose suitable faults based on the state transitions.

1) Node State Extraction: To construct the node state model, we require an effective metric to infer the node state changes during the fault.

Previous fuzzing-based methods [37], [5] use code coverage as the basis for the metric. However, code coverage lacks context and time sequence information to represent the node state within the network. For instance, as shown in Figure 4,

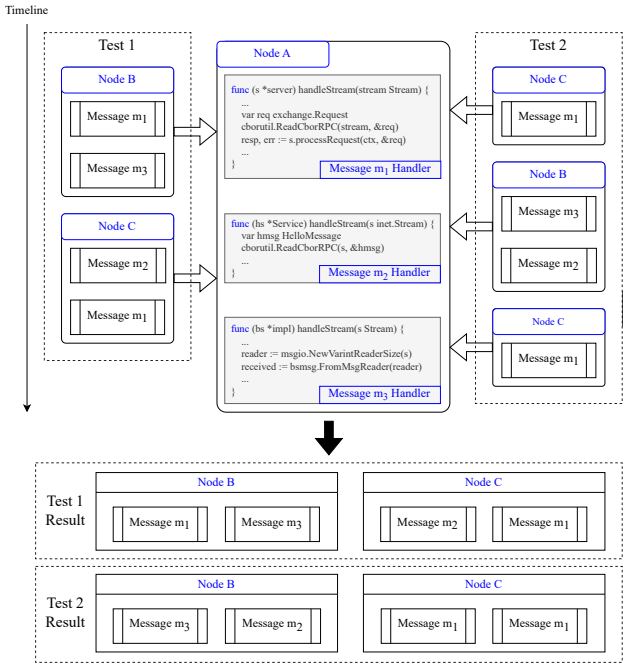


Fig. 4: Example of network message extraction during the test process. The extracted network messages should include context and time sequence information to identify the different node behaviors during test 1 and test 2.

node A can receive three types of messages from remote nodes: (m_1, m_2, m_3). While under test 2, A receives messages in the sequence (B:m_1, B:m_3, C:m_2, C:m_1). While under test 2, A receives messages in the sequence (C:m_1, B:m_3, B:m_2, C:m_1). To capture the message interactions precisely, we should extract the results as receiving messages from node B as (m_1, m_3), and from node C as (m_2, m_1) for test 1. For test 2, the results would be node B as (m_3, m_2) and node C as (m_1, m_1). This way, we can capture the differences in message interactions between nodes B and C during the two tests. However, without information from the network stack, code coverage cannot distinguish these message differences since the corresponding message handler code in node A is called with the same sequence and timing. Thus, it would incorrectly consider tests 1 and 2 as identical.

Other chaos-related methods within distributed databases [41] propose using code instrumentation to extract manually defined node code events as the source of information. However, the blockchain network contains multiple implementations of nodes, not limited to a single programming language. For example, the Ethereum consensus chain has six implementations in different programming languages: Golang [18], Rust [58], Node. js [46], Nim [45], and Java [50]. Similarly, the Filecoin blockchain has two implementations in the same Golang programming language: Lotus [55] and Venus [13]. Thus, it might require significant human effort to define a uniform event, not only across different implementations but also across different programming languages.

Thus, we adopt the network messages between nodes as the data source to ensure a unique information metric across different implementations. Each blockchain has a standardized requirement for the network message format used by all nodes, regardless of their implementation. For instance, the Ethereum beacon chain specifications detail the network message format and the corresponding protocol used for communication between nodes [8]. However, we cannot directly intercept network messages at the system level, unlike previous methods [14], [36]. This is because current blockchain nodes adopt end-to-end traffic encryption, such as Noise [51]. We can only obtain the ciphertext, which is difficult to decode due to the encryption algorithms' resistance to man-in-the-middle attacks [2], and each connection uses a unique key. To overcome this issue, we chose to inject code into the network library used in the P2P network layer to extract network messages. Many current blockchain implementations, such as the Ethereum Beacon Chain [7], Filecoin [12], and Polkadot [53], use the same P2P network framework, libp2p [30], to build their networks. Therefore, we can inject code into this framework to extract the required messages without manually injecting code into every node implementation. For each node's traffic, we extract the message type, sender, receiver, direction, and timestamp as the metrics.

2) State Model Construction: In the state model construction, we aim to build an effective state model that can infer individual node state transitions during faults without being interrupted by other nodes executing different faults.

However, existing methods fail to provide such an effective state model. The model used in CHAOSETH [76] is overly sensitive and can be easily disrupted by other nodes under test. For example, if a target node is receiving new block numbers and another miner is injected with self-mining faults, this miner will broadcast a large number of blocks to the target node, even if the target node is not under test. Mallory [41] adopts a model that is too coarse to infer state transitions during test cases. It uses the entire network event timeline as the state model, requiring events within the distributed database, such as store or load, to be in a fixed sequence and combination. Additionally, it only captures system-level state transfer. Given that we will have multiple faults executing, the state model can not be fine-grained enough to pinpoint which fault triggered a new state within a specific node.

Considering that a node in the blockchain network can be viewed as a finite state machine (FSM) [37], [14], where internal states and inputs determine outputs, we model the node as a Mealy machine [59]. This allows us to capture the node's internal state from both its inputs and outputs. The Mealy machine is formally defined as a tuple $M = (I, O, Q, \delta, \lambda)$. The I and O are finite sets of symbols representing the input and output protocols, respectively. Q denotes the node's internal states, including the initial state q_0 at startup. $\delta : Q \times I \to Q$ is the transition function that maps a state and an input symbol to a new state. This function defines the node's internal state transitions based on the received message and its current state. $\lambda : Q \times I \to O$ is an output function that maps a state and an input symbol to an output symbol. This function defines the node's output based on the current state and the input.

Algorithm 3: Construct the node state model based on network messages.

```
Data: MessageList, Timeline
Result: nodeModel
nodeModel = NewMealyMachine();
alphaList = alphaList.init();
for message ∈ MessageList do
    nodes = message.Group();
    inputProtocol = set();
    outputProtocol = set();
    for node ∈ nodes do
        if node ∈ Timeline.Faults() then
            node.Clear();
        if node.OutputProtocol != Null then
            inputProtocol.Add(node.InputPrtocol);
            outputProtocol.Add(node.OutputPrtocol);
    inputSymbol = alphaList.Map(inputProtocol);
    outputSymbol = alphaList.Map(outputProtocol);
    nodeModel.Learn(inputSymbol, outputSymbol);
```

The Algorithm 3 shows the steps to construct the node Mealy machine. We first collect and extract network messages at 100ms intervals. We then group each message by its target and the corresponding protocols. To exclude state disruptions from nodes undergoing faults that can result in interruptions and inaccuracies in the state transfer model, we first identify all nodes involved in other executing faults from the global fault timeline. Then we filter the message logs based on node identities.

To model the state with a finite alphabet, we map the messages extracted from the node into a finite set of states. For each interval, we group all protocols, treating remote-initiated protocols as inputs and local node-initiated protocols as outputs. Given that a node can have multiple connections to other nodes simultaneously, we infer the node's uniform input and output symbols from different connections by grouping all input protocols as the input symbol and all output protocols as the output symbol. The set of all input protocols is mapped to I, and the set of all output protocols is mapped to O. Since the number of network protocols is limited within each blockchain network to n, the sizes of I and O are also limited to n, making the maximum state space 2^n. Since the state model is used to trace the node state transitions during the fault period, we do not need to construct the node's complete state model. Instead, we focus only on the essential parts where the node's state transitions occur. We then feed the input and output symbols into LearnLib [26] to construct the Mealy machine M.

3) New Fault Selection: After extracting the node state model as feedback, we use it to guide the selection of new faults to inject into the blockchain system. Due to generalization constraints, additional Byzantine faults (e.g., those related to the EVM) may not be suitable for all blockchain systems. For instance, Filecoin does not natively support the EVM; therefore, we select only three types of faults, as shown in Table I. Nevertheless, our approach enables the random combination of crash and Byzantine faults by simultaneously injecting different faults into nodes, resulting in new fault behaviors.

TABLE I: The crash and Byzantine faults that can be injected within the blockchain system.

Fault Type	Fault Name	Description
Crash	Clock skew	System clock adjust
	Offline	Node unavailable
	Network delay	Connection delay
	Packet loss	Packet lost
Byzantine	Corrupted message	Re-order, duplicated or modify packets
	Double-spending	Transactions with the same nonce
	Transaction flooding	Massive normal transactions

Thus, we apply reinforcement learning to select suitable faults for nodes to trigger complex node behavior. In practice, we select Q-learning [69] due to its model-free reinforcement learning to dynamically select novel faults based on observed feedback. The Q-learning is defined by a tuple $(S, A, R, \alpha, \lambda)$. Where S is the set of all the states of the node, A is the set of all the possible faults that can be injected towards the node. The details of each type of fault are shown in Table I. R is the reward function $R(s, a, s') \rightarrow r_i$ that returns a reward received after transitioning from state s to s' under action a. $\alpha \in (0, 1]$ is the learning rate that determines the extent to which newly acquired information overrides old information. $\lambda \in (0, 1]$ is the discount factor that models the importance of future rewards. In Q-learning, the goal is to learn a policy π that maximizes the long-term total reward. The algorithm updates the Q-values based on the equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[R(s,a,s') + \lambda \max_{a'} Q(s',a') - Q(s,a)] \quad (7)$$

Here $Q(s, a)$ represents the value of taking action a in state s. This update rule is applied whenever an agent transitions from state s to s' by taking action a and receiving reward r. The $\max Q(s', a')$ term represents the estimate of the optimal future value, and the learning process adjusts the Q-values towards these long-term rewards.

The state S of the node is computed from the node, the constructed Mealy machine M as a Directed Graph $G = (V, E)$ of the node, where V represents a vertex from each state Q of M and E represents edges as a state transition. Specifically, if the state transition function δ defines a transition from state q to state q', create a directed edge from vertex s to vertex s'. The rewards of the state transfer are computed as a new edge compared with the common edges.

$$R(s_i, a_i, s_{i+1}) = \frac{s_{i+1}.edges - s_i.edges \cap s_{i+1}.edges}{s_i.edges \cap s_{i+1}.edges} \quad (8)$$

To select action a, we adopt the ϵ-greedy Strategy with the following formula.

$$a = \begin{cases} \text{a random action from } A & \text{with probability } \epsilon \\ \max\limits_{a} Q(s,a) & \text{with probability } 1-\epsilon \end{cases} \quad (9)$$

This balances exploration, which avoids the risk of missing out on potentially better actions not yet taken frequently enough to yield an accurate Q-value estimate, and Exploitation, which leverages accumulated knowledge to make the best decision based on current Q-value estimates.

C. Steady State Checker

In the steady state check, we evaluate whether the injected faults have prevented the node from recovering to its normal state. For each node that completes a fault test, we apply our checker in two aspects: functionality and consensus consistency. The functionality check ensures the node remains operational and continues to fulfill its functions, such as providing RPC services. We verify that the node continues to respond to requests and perform expected operations without errors. As the consensus consistency check, we verify that the node's data is consistent with the consensus rules. We ensure the node follows the same chain and maintains the same chain head as other nodes in the network. Additionally, we check that all address balances are consistent across the network. If these checks fail, we conclude that our faults have disrupted the node's steady state. This thorough evaluation helps us identify the impact of injected faults and ensure the robustness of the blockchain system.

V. Evaluation

To evaluate the effectiveness and efficiency of our ChaosChain framework in modeling node state transitions, accelerating the number and types of faults within the blockchain system, and detecting new robustness issues, we address the following research questions:

RQ1: Can ChaosChain effectively model node state?
RQ2: Can ChaosChain achieve efficient fault injection?
RQ3: Can ChaosChain cover more node states than existing methods?
RQ4: Can ChaosChain discover new robustness issues within the blockchain system?
RQ5: How does ChaosChain identify robustness issues in practice?

In Section V-A, we answer RQ1 by evaluating whether our proposed next fault guidance can provide effective and consistent node state transitions during chaos engineering. In Section V-B, we address RQ2 by testing the efficiency of the fault injection approach within Parallel Fault Coordination for injecting faults into the blockchain system. In Section V-C, we examine RQ3 to determine whether our method can cover more node states than existing methods. In Section V-D, we assess ChaosChain's ability to identify new robustness issues within the blockchain system to answer RQ4. In Section V-E, we present a case study to illustrate how ChaosChain identifies robustness issues in real-world scenarios.

We implement and evaluate ChaosChain in Filecoin and Ethereum. We selected Ethereum due to its popularity, and Filecoin because it is based on IPFS and features multiple implementations in various programming languages (Golang and Rust), as well as multiple implementations in the same language (Lotus and Venus in Golang). This is suitable for evaluating our framework's generalizability and effectiveness. On the Filecoin blockchain, since only Lotus nodes are allowed as miner nodes in the local development network, we set up Lotus as the miner node. Specifically, we configure the network with three Lotus miner nodes, three Lotus validator nodes, three Venus validator nodes, and three Forest (Rust) validator nodes. For Ethereum, we set up the Golang implementation Prysm [56] with three validator nodes. We use Kubernetes [16] to run each node and inject the corresponding faults. The crash faults are conducted by using ChaosMesh [15]. The experiments are conducted on an Ubuntu 20.04 system with a 128-core CPU and 512GB of memory.

We compare ChaosChain with state-of-the-art chaos tools, specifically AttackNet [47], developed by Trail of Bits and the Ethereum Foundation for Ethereum. We adapted AttackNet for use with Filecoin while maintaining the same configuration. Unfortunately, our comparison could not include Phoenix [38] because its source code is not fully released. ChaosETH [76] is specifically developed for a single-node implementation of the Ethereum execution chain, which is gradually being replaced by the Ethereum beacon chain. Migrating ChaosETH to other implementations requires significant effort because manual syscall interception is required, which may follow patterns different from those of the Ethereum execution chain. Mallory [41] is designed for distributed databases where it is possible to extract uniform code events and requires fixed database execution. However, as explained in the previous section, finding a uniform code event as the data source and generating uniform actions, such as database operations, is challenging. Additionally, encrypted connections make it difficult to capture network traffic at the system level.

A. Effectiveness of Node State Model (RQ1)

This section aims to evaluate whether our node state model can effectively capture the node state transitions during fault execution. We assess the on the local Filecoin blockchain system, which includes two Golang implementations (Lotus and Venus) and one Rust implementation (Forest).

To demonstrate that the node state model performs consistently across different nodes, we first run ChaosChain without injecting faults to evaluate the effectiveness of our model across different libp2p implementations. We set up the blockchain with each implementation running on a separate node to avoid interference. After running for 12 hours, we collect the final state model of each node and compare the common edges across node state models to evaluate their similarity. The results are shown in Table II. Nodes of the same type exhibit a higher common edge rate, indicating that our node state model can consistently reproduce node state transitions. Furthermore, we observe that nodes from different implementations exhibit lower state-model similarity, suggesting that they follow distinct processing logic for network messages. This finding implies that it may be possible to identify the node type based on its network message patterns.

To demonstrate that our node state model can accurately extract node state transitions during fault execution, we run ChaosChain with the same setup as above, but we enable fault injections this time. We use a fixed timeline and disable the guidance to ensure that both runs adopt the same fault injection sequence. The results, shown in Table III, are similar to the results without fault execution, indicating that our state model can consistently reproduce node state transitions even with fault execution. The reason both the Lotus miner and

TABLE II: The result of the common edge percentage of node state models without faults, M stands for miner node and V stands for validator node.

Node Name	lotus-M-1	lotus-V-1	forest-V-1	venus-V-1
lotus-M-0	0.89	0.67	0.22	0.27
lotus-V-0	0.66	0.84	0.13	0.25
forest-V-0	0.33	0.12	1.00	0.50
venus-V-0	0.33	0.37	0.50	1.00

TABLE III: The result of the common edge percentage of node state models with faults, M stands for miner node and V stands for validator node.

Node Name	lotus-M-1	lotus-V-1	forest-V-1	venus-V-1
lotus-M-0	0.99	0.92	0.51	0.33
lotus-V-0	0.86	0.97	0.50	0.46
forest-V-0	0.53	0.55	1.00	0.33
venus-V-0	0.42	0.51	0.43	0.93

Lotus validator nodes have higher similarity is that both node types use the same implementation. Consequently, they behave similarly under the same conditions.

Finally, to evaluate the effectiveness of the timeline in filtering out interference messages from other nodes, we ran ChaosChain without the disrupted message filtration while keeping all other settings the same as described above. The results are shown in Table IV, where all parameters drop significantly. By comparing these results with those obtained using the timeline message filter, we can assess the effectiveness of the disrupted message filtration in removing interference messages from nodes under fault.

Answer to RQ1: Our node state model can effectively capture the node state transitions within the blockchain system, both with and without faults, and can remove interference messages from other nodes under different faults.

B. Efficiency of Fault Injection (RQ2)

To evaluate the efficiency of our framework in accelerating the number of faults during chaos engineering, we compare the number of faults injected into the blockchain system. We run our ChaosChain and attacknet with the same configuration for 12 hours on the local Filecoin blockchain system.

The results are shown in Figure 5. Our method successfully injected 786 faults into the blockchain system across all nodes during the test period, whereas Attacknet only injected 215, achieving approximately a threefold improvement. In the early stage, our method injected faults more broadly by first conducting an exploration process, injecting diverse faults to collect feedback from different nodes within the system. The gradual rise in the curve indicates that our reinforcement learning method has accumulated sufficient knowledge to select appropriate faults for the system. Furthermore, our parallel fault coordination mechanism efficiently maximizes the number of faults injected into the blockchain system.

TABLE IV: The result of the common edge percentage of node state models with faults without timeline message filter, M stands for miner node and V stands for validator node.

Node Name	lotus-M-1	lotus-V-1	forest-V-1	venus-V-1
lotus-M-0	0.45	0.14	0.01	0.02
lotus-V-0	0.17	0.23	0.04	0.04
forest-V-0	0.08	0.01	0.33	0.30
venus-V-0	0.14	0.14	0.17	0.29

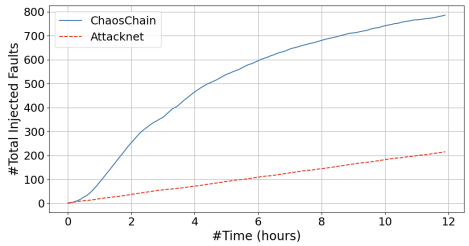


Fig. 5: The total number of faults that are injected into the blockchain system.

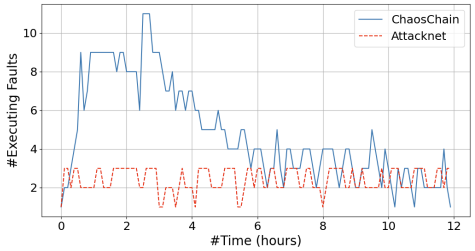


Fig. 6: The number of faults that are injected into the blockchain system simultaneously.

In addition, we also analyze the maximum number of faults in the system at the same time. As shown in Figure 6, our ChaosChain has a maximum of twelve faults executing within the blockchain, while the Attacknet has about three faults. This demonstrates that our parallel fault coordination can inject as many faults as possible to speed up the chaos engineering process while maintaining the blockchain consensus's security constraints.

Answer to RQ2: Our fault injection can efficiently improve the fault numbers during chaos engineering within the blockchain system by injecting three times more faults in total and four times more faults simultaneously.

C. State Coverage of ChaosChain (RQ3)

In this section, we measure the node state coverage explored by our ChaosChain framework within each implementation and compare it with two existing methods, Attacknet and ChaosMesh [15]. We extend our evaluation to include all implementations within Ethereum (Prysm) and Filecoin (Lotus, Forest, and Venus).

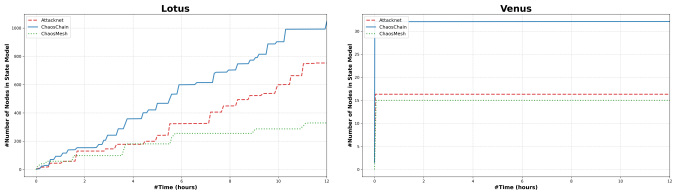
The results shown in Figure 7 demonstrate that ChaosChain covers more states than Attacknet and ChaosMesh within the 12-hour evaluation across all three Filecoin implementations and Ethereum's Prysm implementation.

At the start of each experiment, ChaosChain, Attacknet, and ChaosMesh exhibit similar node state coverage, though ChaosChain is slightly lower than the other two methods.

This is because the Q-learning method begins by learning the feedback from each fault on each node. Initially, it sequences single faults toward the nodes to obtain feedback on node state transitions, training its fault selection policy with accumulated knowledge rather than focusing on exploring new states.

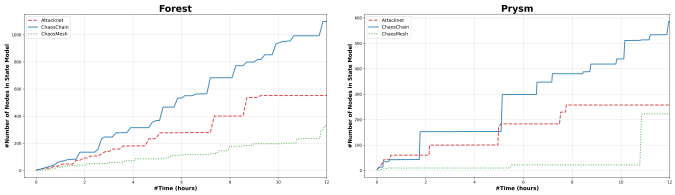
As testing progresses, our reinforcement learning method, Q-learning, accumulates sufficient information within the first four hours and begins to explore new faults that can trigger complex node transitions. Consequently, we observe that node states become increasingly complex over time, indicating the effectiveness of ChaosChain in uncovering intricate node behaviors. Specifically, in the Lotus implementation, our framework covers 1,047 node states, while Attacknet covers 754 and ChaosMesh covers 329. In the Venus implementation, ChaosChain, Attacknet, and ChaosMesh cover 32, 17, and 15 node states, respectively. For the Forest implementation, ChaosChain covers nearly twice as many states as Attacknet, with 1,096 compared to 551, and approximately triple those of ChaosMesh at 338. In the Ethereum Prysm implementation, ChaosChain covers over twice as many states as Attacknet (585) and ChaosMesh (223).

Venus [13] exhibits notably lower state coverage than the other implementations because it is a simplified Filecoin implementation that provides only basic data query services on the chain. Therefore, it does not support all the functionalities required of a full node within Filecoin, such as block mining and message storage, unlike Lotus, the official implementation of Filecoin. As shown in the state coverage results in Table V, ChaosChain, Attacknet, and ChaosMesh all achieve significantly lower state coverage in Venus compared to Lotus. This limitation reduces the diversity of node behaviors in response to fault scenarios and different responses from remote nodes.



(a) Filecoin Golang implementation Lotus

(b) Filecoin Golang implementation Venus



(c) Filecoin Rust implementation Forest

(d) Ethereum Golang implementation Prysm

Fig. 7: The number of nodes within each node state model across implementations of Filecoin and Ethereum.

In addition to the number of nodes in the state model, we compare the code coverage of ChaosChain, Attacknet, and ChaosMesh. Since Golang only supports statement coverage, for the Golang implementations Lotus and Venus, we collect statement coverage via the default Go method [40]. For the Rust implementation Forest, to maintain consistency with the Golang implementations, we adopt line coverage via LLVM [34]. We set up the same local network and collected data over 12 hours. As shown in Table V, our framework achieves up to 20% higher statement coverage than Attacknet and up to 45% higher statement coverage than ChaosMesh. This demonstrates the effectiveness of next-fault guidance and parallel fault coordination in generating suitable faults and executing them concurrently within the system to cover a broader node code space.

TABLE V: The statement coverage of all the implementations of the blockchain system.

Blockchain	Implementation	ChaosChain	Attacknet	ChaosMesh
Filecoin	Lotus	15,155	12,578	10,425
	Venus	8,772	8,015	7,904
	Forest	92,441	78,244	70,864
Ethereum	Prysm	21,022	19,704	18,371

Answer to RQ3: Our ChaosChain covers over twice as many node states and achieves at least 6.7% and at most 45% more code coverage than Attacknet and ChaosMesh, with an average of 13% over Attacknet and 24% over ChaosMesh.

D. Capability of Detecting the Robustness Issues (RQ4)

To evaluate ChaosChain's ability to detect robustness issues, we run it on all implementations of both Ethereum and Filecoin, as described in RQ3. As shown in Table VI, ChaosChain detected six previously unknown robustness issues: three in Lotus, two in Venus, and one in Forest. Two of these issues (#1 and #2) have been confirmed by the developers. Two issues (#2 and #6) are detected by the function state checker, causing nodes to crash and potentially enabling DoS attacks. The consensus state checker detects four issues, leading nodes to fail to sync with the latest blockchain state. To analyze false positives, we manually reproduced the chaos engineering process by replaying the Lamport timeline and did not find any false positives. This confirms the accuracy and effectiveness of ChaosChain in identifying robustness issues within blockchain systems.

Answer to RQ4: Our ChaosChain can trigger and detect complex issues within the blockchain system and P2P network infrastructure by identifying six new robustness issues.

E. Case Study for Identifying Robustness Issues

We present a case study demonstrating how ChaosChain can identify issues not only in the implementation but also in libp2p's infrastructure, as seen in Issue #1. This issue prevented the node from sending a valid message to a remote node, thereby preventing it from fetching missing blocks from the blockchain. Although it appears similar to the motivation described in Section III-B, it is caused by an opposite issue.

TABLE VI: STATISTICS OF NEW ROBUSTNESS ISSUES DETECTED BY CHAOSCHAIN.

ID	Implementation	Robustness Issue	Description	Bug State
1	Lotus	Consensus disruption	Node sending message queue fills up due to chaos, preventing it from requesting missing blocks.	Confirmed
2	Lotus	Function disruption	Node has memory leaks due to uncanceled goroutines.	Confirmed
3	Forest	Consensus disruption	Node considers the new block from the miner under chaos as invalid.	Confirmed
4	Venus	Consensus disruption	Node considers the latest block from the miner under chaos as invalid.	Investigating
5	Venus	Consensus disruption	Node can not fetch the missing block after corrupted message and clock skew faults.	Confirmed
6	Lotus	Function disruption	Node crashes due to SIGSEGV when injected with both transaction flooding and clock skew faults.	Investigating

```
func (s *Stream) write() {
    ...
    window := atomic.LoadUint32(&s.sendWindow)
    if window == 0 {
        select {
        case <-s.sendNotifyCh:
            goto START
        case <-s.writeDeadline.wait():
            return 0, ErrTimeout
        }
    }
    ...
    max = min(window, MaxMessageSize, uint32(len(b)))
    s.session.sendMsg(b[:max])
    atomic.AddUint32(&s.sendWindow, ^uint32(max-1))
    ...
}
func (s *Session) handleStreamMessage() {
    ...
    if hdr.MsgType() == typeWindowUpdate {
        stream.incrSendWindow(hdr, flags)
    }
    ...
}
```

Fig. 8: Simplified code snippet demonstrating the consensus disruption issues in Lotus.

In the libp2p framework, each node communicates with others through streams that employ a send/receive window mechanism similar to TCP [72]. As illustrated in Figure 8, when sending a message to a remote node via the *handleSendingMessages* function, the *stream.write* function first checks whether the send window size is greater than 0. If so, the message can be sent immediately. However, if the window size reaches 0, the sender must wait until it increases before proceeding.

The issue arises because the window size update depends on messages from the remote node. Consequently, the write function must wait for a window-size update message from the remote node. When this update message is dropped due to injected network packet-drop chaos, the write function becomes indefinitely blocked because libp2p does not account for this scenario and lacks a timeout.

Although both implementations adopt the same cache channel mechanism for sending large messages, their handling strategies differ significantly. The Golang implementation follows a "whole message first" approach—it defers sending the window update message until the window size is large enough to accommodate the entire message. In contrast, the JavaScript implementation prioritizes immediate transmission by chunking the message: it first sends data up to the current window size, then waits for a window size update to transmit the remaining portions. Additionally, the JavaScript version uses a queue that prioritizes the most recent messages.

Once the initial window size update message is dropped, the js-libp2p node begins sending partial fragments of new blocks without properly completing the transmission of the previous large block. If large blocks continue to arrive, other nodes will persistently receive incomplete fragments of only the latest blocks, with corrupted sequencing and data. As a result, they can never reconstruct a complete, valid block from the JavaScript node, ultimately leading to severe synchronization failures across the network.

Though both the Golang and JavaScript implementations set the initial window size to a relatively small value of 256KB. Considering that the average Ethereum block size is below 256KB, this configuration is generally sufficient under normal conditions. However, on January 6, 2024 [11], the maximum block size reached 272KB, exceeding the window size limit and inevitably triggering the need for a window update message. Such real-world conditions can trigger these issues, leading to the aforementioned synchronization failures in production environments.

VI. DISCUSSION

A. Fault Types Can Be Injected Into the Blockchain Systems

Currently, our framework supports crash faults from Chaos-Mesh [15] and three Byzantine faults: corrupted connection, double-spending, and transaction flooding, which are common across all blockchain systems due to the fundamental transfer and mining functions. By combining crash and Byzantine faults, we can produce more complex fault scenarios, such as simulating corrupted transactions through transaction faults combined with network packet drops. However, additional Byzantine faults, such as those related to the Ethereum Virtual Machine (EVM) [42], are more specific to certain blockchain systems, such as Ethereum. Other blockchain systems, such as Filecoin [12], do not natively support EVM, making these faults unsuitable for them. Therefore, we plan to expand our framework by adding more Byzantine faults specific to each blockchain system to simulate more realistic, complex fault events. This will enhance our framework's ability to handle diverse and nuanced fault scenarios across different blockchain environments.

B. Generality to Other Blockchain Systems

Our framework is designed to inject code into the peer-to-peer network framework to extract network messages for state model construction, thereby ensuring generalizability rather than relying on system-specific implementations. In

this work, we primarily focus on the libp2p framework [30]. Consequently, any blockchain system that adopts libp2p for its peer-to-peer networking can be supported by our framework by injecting modified code into libp2p to extract network messages. Moreover, our framework can be readily adapted to other peer-to-peer networks, to light and mobile nodes that share the same network architecture, and to Layer 2 networks.

For example, in addition to libp2p, there is another peer-to-peer network framework called devp2p [6], which is used by the Ethereum Execution chain [9]. The devp2p framework is specifically designed to meet the needs of the Ethereum execution network. It is not widely used, as Ethereum migrates from devp2p to libp2p [52]. Nonetheless, our framework can also be applied to devp2p or other blockchain systems with self-developed peer-to-peer networks by adding around 100 lines of code to extract the network messages.

As for the steady-state checker, our framework requires developers to provide suitable RPC addresses and methods to check both the node state and the consensus state. Various blockchain systems commonly use RPC for fetching node states and provide sufficient documentation for developers to obtain [29]. Therefore, developers only need to modify the method to specify which data needs to be consistent across different nodes.

C. Guidance Method Based on the Node State

Our ChaosChain uses the Q-learning reinforcement learning method [69] to select suitable faults based on extracted node transition information from the node Mealy state machine. Due to Q-learning's model-free nature and simplicity, it does not require an environment model. It learns directly from interactions with the environment, reducing the manual effort to model the blockchain environment.

However, our guidance method can be replaced with more complex reinforcement learning methods, such as Deep Reinforcement Learning [24], which combines reinforcement learning with deep learning techniques to handle high-dimensional state and action spaces. Other guidance methods, such as mutation-based guidance, could also be used [19]. We can generate new fault sequences to inject into the blockchain system by mutating previously effective ones based on the node state model.

By adopting these methods, we can change fault types and sequences, and adjust fault parameters such as duration and chaos intensity, by combining them with additional information. This allows us to explore more complex node behavior within the blockchain system.

D. Security Constraint Type Selection

Our four security constraints are designed to be necessary and sufficient for chaos engineering purposes, rather than comprehensive attack modeling. Although existing blockchain systems have additional interdependencies—such as the Expected Consensus used by Filecoin [12], which allows zero validators per epoch—we designed these security constraints to ensure generalizability across different blockchain implementations. This prevents general faults from breaking the system (e.g., eclipse attacks [23]), which would produce meaningless test results. Our framework supports extensibility for system-specific constraints once the specifications for a given blockchain system are defined. Our evaluation demonstrates the effectiveness of these constraints in practice.

E. Ablation Study on Security Constraints

The security constraints are designed to prevent fault injections that would violate the consensus requirements of the blockchain system. Removing these constraints would increase the fault injection rate, since faults could be introduced without restriction. However, such unrestricted fault injection would produce misleading rather than meaningful results: the resulting node misbehavior would stem from deliberate violations of the consensus protocol's security assumptions, not from genuine robustness deficiencies. In other words, the observed failures would arise from operating outside the protocol's valid threat model, making it impossible to draw reliable conclusions about system robustness. Therefore, we consider an ablation study that removes the security constraints to be inappropriate.

VII. Related Work

A. Chaos Engineering in Distributed Systems

Chaos engineering has been widely used within distributed systems to evaluate software system reliability.

Chaos Monkey [43] is the first method to randomly shut down pre-defined servers to test system resilience, designed explicitly for Netflix server scenarios. Jepsen [27] is a chaos engineering tool for distributed databases that injects crash faults into the system. It requires manual configuration of fault types, sequences, and target nodes before starting the chaos engineering process. Since Jepsen executes illegal distributed database operations to check the system state, it cannot inject Byzantine faults. Mallory [41] improves upon Jepsen by adding a feedback function with guidance to automate the selection of new faults to inject into the system. However, its method is only suitable for coarse-grain system state feedback rather than fine-grain extraction of single node state transitions. Additionally, its feedback function requires developers to manually instruct the source code on which events to extract, rather than automatically extracting uniform network messages. As for blockchain systems, CHAOSETH [76] is developed to evaluate the robustness of single nodes under chaos induced by system calls. It analyzes the Ethereum source code to capture the Ethereum node system call pattern for fault injection and uses built-in metrics to evaluate the steady state. This makes migrating to other blockchain systems difficult, as these implementations have different system call patterns and built-in metrics. Additionally, it focuses on the stability of single node implementations in local environments rather than the robustness of the entire blockchain system, including handling local environment issues and Byzantine faults from other nodes. Attacknet [25] is similar to Jepsen, requiring manual configuration of fault types, sequences, and target nodes. Although it is designed for blockchain systems, it can only inject crash faults rather than Byzantine faults. This limitation

makes it difficult to generate complex fault types within the blockchain system. While Phoenix [38] applies chaos engineering to detect resilience issues in blockchain implementations, their method focuses only on injecting Byzantine faults. Additionally, their approach relies on manually pre-defined nodes for fault injection and requires manual definition of fault injection rules. Phoenix's fault coordination only allows a single fault within the blockchain system at a time. For generalizability, their work requires manual modifications to the source code for each implementation, making it labor-intensive to migrate to other implementations or programming languages.

All the above methods lack a feedback function to select the most suitable faults for the blockchain system based on previous node results. Additionally, their fault injection approaches support only one fault type at a time. They cannot inject multiple types of faults into a single node to simulate real-world complex system behavior.

Main difference. Our framework identifies robustness issues related to blockchain system infrastructures by injecting both Byzantine and crash faults. It aims to be generally applicable across nodes in the blockchain system with various implementations and programming languages. This comprehensive approach ensures a broader, more robust evaluation across different blockchain environments and implementations.

B. Blockchain Fuzzing

Current blockchain fuzzing research primarily concentrates on targeting blockchain consensus mechanisms [38], [5], [37], [74], [29], [22], [60] and smart contracts [21], [28], [77], [73], [44], [75]. Smart contracts are built on blockchain consensus but are not directly related to network operations and thus fall outside our scope, which is focused on blockchain network layer infrastructure.

For instance, EtherDiffer [29] primarily targets Ethereum Node RPC servers rather than the peer-to-peer (P2P) network. Similarly, blockchain consensus fuzzing tools such as Tyr [5], Loki [37], Fluffy [74], and Chronos [4] target different blockchain consensus protocols using fuzzing techniques. Fluffy [74] evaluates the differential behavior of the EVM via mutation transactions. Tyr [5] and Loki [37] select some fixed nodes within the blockchain system to broadcast Byzantine fault messages via fuzzing. Chronos [4] focuses on node implementation bugs related to timeout processing.

Main difference. Our framework identifies robustness issues related to blockchain system infrastructures by injecting both Byzantine and crash faults. It is designed to be applicable across nodes in the blockchain system with various implementations and programming languages. Unlike previous fuzzing works, which primarily inject Byzantine faults specific to certain blockchain systems and focus mainly on particular implementations or programming languages, our framework aims to provide a more comprehensive and adaptable approach. This allows for a broader, more robust evaluation across different blockchain environments and implementations.

VIII. Conclusion

In this work, we present ChaosChain, a framework for blockchain systems that enables effective, efficient chaos engineering to trigger and identify robustness issues.

By introducing parallel fault coordination and next fault guidance, ChaosChain enhances the robustness evaluation of blockchain networks. In parallel fault coordination, we construct a global timeline of all executing faults and enforce security constraints to comply with the blockchain consensus protocol. This allows us to inject multiple fault types into an arbitrary number of nodes simultaneously. In the next fault guidance, we propose adopting the node state model as a Mealy state model to provide effective feedback for reinforcement learning. This helps in selecting suitable faults to introduce into the chaos engineering process.

We evaluated our framework across four Filecoin and Ethereum blockchain system implementations, detecting six new robustness issues. To evaluate the efficiency of parallel fault coordination and next fault guidance, we compared our framework with state-of-the-practice tools, such as Attacknet and ChaosMesh. The results demonstrate that our framework can execute four times as many faults simultaneously, achieving an average code coverage of over 13% higher than that of existing methods.

IX. Acknowledgments

We would like to thank the anonymous reviewers and COMPASS members for their insightful comments. This work is partly supported by the National Natural Science Foundation of China under Grant No.U2541211, No.62372218, and No.U24A6009, as well as Hong Kong RGC Projects PolyU15224121 and PolyU15231223.

References

[1] A. Basiri, N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, 2016.

[2] F. Callegati, W. Cerroni, and M. Ramilli, "Man-in-the-middle attack to the https protocol," *IEEE Security & Privacy*, 2009.

[3] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OsDI*, 1999.

[4] Y. Chen, "Chronos: Finding timeout bugs in practical distributed systems by deep-priority fuzzing with transient delay," in *2024 IEEE Symposium on Security and Privacy (SP)*, 2024.

[5] Y. Chen, F. Ma, Y. Zhou, Y. Jiang, T. Chen, and J. Sun, "Tyr: Finding consensus failure bugs in blockchain system with behaviour divergent model," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[6] Ethereum, "Devp2p," https://github.com/ethereum/devp2p, 2024.

[7] ——, "Ethereum," https://ethereum.org/en/, 2024.

[8] ——, "Ethereum Proof-of-Stake Consensus Specifications," https://github.com/ethereum/consensus-specs, 2024.

[9] ——, "Go Ethereum," https://github.com/ethereum/go-ethereum, 2024.

[10] ethernodes.org, "Ethereum Mainnet Statistics," https://ethernodes.org/, 2024.

[11] Etherscan, "Ethereum Average Block Size Chart," https://etherscan.io/chart/blocksize, 2026.

[12] Filecoin, "Filecoin," https://filecoin.io/, 2024.

[13] filecoin project, "Venus," https://github.com/filecoin-project/venus, 2024.

[14] P. Fiterau-Brostean, B. Jonsson, K. Sagonas, and F. Tåquist, "Automata-based automated detection of state machine bugs in protocol implementations." in *NDSS*, 2023.

[15] C. N. C. Foundation, "Chaos Mesh," https://github.com/chaos-mesh/chaos-mesh, 2024.

[16] ——, "Kubernetes," https://kubernetes.io/, 2024.

[17] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016.

[18] golang, "The Go Programming Language," https://github.com/golang/go, 2024.

[19] R. Gopinath, P. Görz, and A. Groce, "Mutation analysis: Answering the fuzzing challenge," *arXiv preprint arXiv:2201.11303*, 2022.

[20] Gremlin, "Gremlin," https://www.gremlin.com/docs, 2026.

[21] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: effective, usable, and fast fuzzing for smart contracts," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[22] H. Guo, W. Gan, M. Zhao, C. Zhang, T. Wu, L. Zhu, and J. Xue, "Prichain: Efficient privacy-preserving fine-grained redactable blockchains in decentralized settings," *Chinese Journal of Electronics*, vol. 34, no. 1, pp. 82–97, 2025.

[23] E. Heilman, A. Kendler, A. Zohar, and S. Goldberg, "Eclipse attacks on {Bitcoin's}{peer-to-peer} network," in *24th USENIX security symposium (USENIX security 15)*, 2015.

[24] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," in *Proceedings of the AAAI conference on artificial intelligence*, 2018.

[25] https://www.trailofbits.com/, "Attacknet," https://github.com/crytic/attacknet, 2024.

[26] M. Isberner, F. Howar, and B. Steffen, "The open-source learnlib: a framework for active automata learning," in *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I 27*, 2015.

[27] jepsen io, "Jepsen," https://github.com/jepsen-io/jepsen, 2024.

[28] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[29] S. Kim and S. Hwang, "Etherdiffer: Differential testing on rpc services of ethereum nodes," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023.

[30] P. Labs, "Libp2p a modular network stack," https://github.com/libp2p/libp2p, 2024.

[31] L. Lamport, "The part-time parliament," in *Concurrency: the Works of Leslie Lamport*, 2019.

[32] ——, "Time, clocks, and the ordering of events in a distributed system," in *Concurrency: the Works of Leslie Lamport*, 2019.

[33] B. Lashkari and P. Musilek, "A comprehensive review of blockchain consensus mechanisms," *IEEE access*, vol. 9, pp. 43 620–43 652, 2021.

[34] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.*, 2004.

[35] LitmusChaos, "Litmus," https://docs.litmuschaos.io/, 2026.

[36] Z. Luo, J. Yu, F. Zuo, J. Liu, Y. Jiang, T. Chen, A. Roychoudhury, and J. Sun, "Bleem: packet sequence oriented fuzzing for protocol implementations," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[37] F. Ma, Y. Chen, M. Ren, Y. Zhou, Y. Jiang, T. Chen, H. Li, and J. Sun, "Loki: State-aware fuzzing framework for the implementation of blockchain consensus protocols." in *NDSS*, 2023.

[38] F. Ma, Y. Chen, Y. Zhou, J. Sun, Z. Su, Y. Jiang, J. Sun, and H. Li, "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*, 2023.

[39] F. Mattern *et al.*, *Virtual time and global states of distributed systems*. Univ., Department of Computer Science, 1988.

[40] T. McIntosh, "Code coverage for Go integration tests," https://go.dev/blog/integration-test-coverage, 2024.

[41] R. Meng, G. Pîrlea, A. Roychoudhury, and I. Sergey, "Greybox fuzzing of distributed systems," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[42] minimalsm, "Ethereum Virtual Machine (EVM)," https://ethereum.org/en/developers/docs/evm/, 2024.

[43] Netflix, "Chaos Monkey," https://github.com/Netflix/chaosmonkey, 2024.

[44] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020.

[45] nim lang, "Nim," https://github.com/nim-lang/Nim, 2024.

[46] nodejs, "Node.js," https://github.com/nodejs/node, 2024.

[47] T. of Bits, "Trail of Bits," https://www.trailofbits.com/, 2024.

[48] offchainlabs, "Post-Mortem Report: Ethereum Mainnet Finality (05/11/2023)," https://medium.com/offchainlabs/post-mortem-report-ethereum-mainnet-finality-05-11-2023-95e271dfd8b2, 2024.

[49] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX annual technical conference (USENIX ATC 14)*, 2014.

[50] Oracle, "Java," https://www.java.com/, 2024.

[51] T. Perrin, "The Noise Protocol Framework," https://noiseprotocol.org/noise.htmls, 2024.

[52] pettinarip, "Ethereum network layer," https://ethereum.org/en/developers/docs/networking-layer/, 2024.

[53] Polkadot, "Polkadot," https://polkadot.network/, 2024.

[54] potuz, "Ethereum Mainnet Finality Fixed Issues," https://github.com/prysmaticlabs/prysm/pull/12387, 2024.

[55] F. project, "Lotus," https://github.com/filecoin-project/lotus, 2024.

[56] prysmaticlabs, "Prysm: An Ethereum Consensus Implementation Written in Go," https://github.com/prysmaticlabs/prysm, 2024.

[57] M. Raynal, "About logical clocks for distributed systems," *ACM SIGOPS Operating Systems Review*, 1992.

[58] rust lang, "The Rust Programming Language," https://github.com/rust-lang/rust, 2024.

[59] M. Shahbaz and R. Groz, "Inferring mealy machines," in *International Symposium on Formal Methods*, 2009.

[60] D. Shi, X. Wang, M. Xu, L. Kou, and H. Cheng, "Ress: A reliable and effcient storage scheme for bitcoin blockchain based on raptor code," *Chinese Journal of Electronics*, vol. 32, no. 3, pp. 577–586, 2023.

[61] sigp, "Lighthouse: Ethereum consensus client," https://github.com/sigp/lighthouse, 2024.

[62] M. Singhal and A. Kshemkalyani, "An efficient implementation of vector clocks," *Information Processing Letters*, 1992.

[63] T. I. TEAM, "51% Attack: Definition, Who Is At Risk, Example, and Cost," https://www.investopedia.com/terms/1/51-attack.asp, 2024.

[64] TippyFlitsUK, "https://github.com/filecoin-project/lotus/issues/10906," https://github.com/filecoin-project/lotus/issues/10906, 2024.

[65] M. Tran, A. Shenoi, and M. S. Kang, "On the {Routing-Aware} peering against {Network-Eclipse} attacks in bitcoin," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[66] wackerow, "PROOF-OF-STAKE (POS)," https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/, 2024.

[67] ——, "PROOF-OF-WORK (POW)," https://ethereum.org/en/developers/docs/consensus-mechanisms/pow/, 2024.

[68] W. Wang, S. Deng, J. Niu, M. K. Reiter, and Y. Zhang, "Engraft: enclave-guarded raft on byzantine faulty nodes," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[69] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, 1992.

[70] wikipedia, "Chaos engineering," https://en.wikipedia.org/wiki/Chaos_engineering, 2024.

[71] ——, "Double-spending," https://en.wikipedia.org/wiki/Double-spending, 2024.

[72] Wikipedia, "Transmission Control Protocol," https://en.wikipedia.org/wiki/Transmission_Control_Protocol, 2024.

[73] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[74] Y. Yang, T. Kim, and B.-G. Chun, "Finding consensus bugs in ethereum via multi-transaction differential fuzzing," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, 2021.

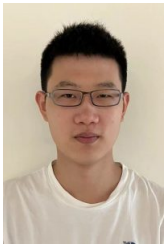
[75] M. Ye, Y. Nan, Z. Zheng, D. Wu, and H. Li, "Detecting state inconsistency bugs in dapps via on-chain transaction replay and fuzzing," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023.

[76] L. Zhang, J. Ron, B. Baudry, and M. Monperrus, "Chaos engineering of ethereum blockchain clients," *Distributed Ledger Technologies: Research and Practice*, 2023.

[77] Q. Zhang, Y. Wang, J. Li, and S. Ma, "Ethploit: From fuzzing to efficient exploit generation against smart contracts," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020.

[78] P. Zheng, Z. Zheng, and L. Chen, "Selecting reliable blockchain peers via hybrid blockchain reliability prediction," *IET Software*, 2023.

[79] zkVlad, "Ethereum client diversity," https://ethereum.org/en/developers/docs/nodes-and-clients/client-diversity, 2024.

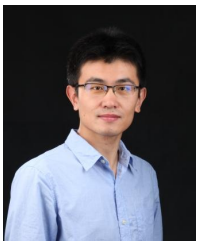


Junjie Ma received the Bachelor degree in Computer Science and Technology from Northeastern University, China in 2022. He is currently working towards the Ph.D. degree under the collaborative Ph.D. program in the Department of Computer Science and Engineering at Southern University of Science and Technology and the Department of Computing at The Hong Kong Polytechnic University, under the supervision of Prof. Daniel Xiapu Luo, Prof. Wang Qi, and Prof. Fengwei Zhang. His current research interests include decentralized autonomous organizations and blockchain security.

Muhui Jiang obtained his Ph.D. degree in Department of Computing from the the Hong Kong Polytechnic University. Before coming to PolyU, He received his B.Eng. in Department of Software Engineering, Tongji University in 2016. His current research interests include blockchain security, network security, system security and IoT security. More specifically, He is interested in reverse engineering, binary analysis, firmware rehosting, fuzzing techniques, and Defi security.

Edmond W.W. Chan received the PhD degree in computer science from the Department of Computing, the Hong Kong Polytechnic University, in 2010. He was a researcher at the Huawei Noah's Ark Laboratory and worked on mobile network performance and traffic analysis. He is currently a principal solutions architect with Akamai Technologies, Hong Kong, addressing performance and security problems in web, media, DNS, and TCP for clients in different verticals including finance, enterprise, gaming, and mobile and consumer electronics.

Xiapu Luo is a professor at the Department of Computing of the Hong Kong Polytechnic University. His research focuses on Blockchain and Smart Contracts Security, Mobile and IoT Security, Network Security and Privacy, and Software Engineering with papers published in top-tier security, software engineering, and networking venues. His research led to more than ten best/distinguished paper awards, including ACM CCS'24 Distinguished Paper Award, three ACM SIGSOFT Distinguished Paper Awards in ICSE'24, ISSTA'22 and ICSE'21, Best DeFi Papers Award 2023, Best Paper Award in INFOCOM'18, Best Research Paper Award in ISSRE'16, etc. and several awards from the industry. He received the BOCHK Science and Technology Innovation Prize (FinTech) for his contribution to blockchain security. He regularly serves in the program committees of top security and software engineering conferences and received Top Reviewer Award from CCS'22 and Distinguished TPC member Award from INFOCOM'23 and INFOCOM'24.



Qi Wang received the Ph.D. degree in computer science and engineering from The Hong Kong University of Science and Technology (HKUST) in 2011. From October 2011 to September 2013, he was an Alexander von Humboldt Post-Doctoral Researcher at Otto von Guericke University Magdeburg, Germany. He has been with the Department of Computer Science and Engineering, Southern University of Science and Technology, since 2014, where he is currently a tenured Associate Professor. His research interests include cryptography and coding theory.

Fengwei Zhang received the Ph.D. degree in computer science from George Mason University. He is currently an Associate Professor with the Department of Computer Science and Engineering, Southern University of Science and Technology (SUSTech). His research interests include systems security, with a focus on trustworthy execution, hardware-assisted security, debugging transparency, transportation security, and plausible deniability encryption.