Phoenix: Detect and Locate Resilience Issues in Blockchain via Context-Sensitive Chaos

Fuchen Ma
BNRist, Tsinghua University
Beijing, China

Yuanliang Chen*
BNRist, Tsinghua University
Beijing, China

Yuanhang Zhou
BNRist, Tsinghua University
Beijing, China

Jingxuan Sun
Beijing University of Posts and Telecommunications
Beijing, China

Zhuo Su
BNRist, Tsinghua University
Beijing, China

Yu Jiang†
BNRist, Tsinghua University
Beijing, China

Jiaguang Sun
BNRist, Tsinghua University
Beijing, China

Huizhong Li
WeBank
Shenzhen, China

ABSTRACT

Resilience is vital to blockchain systems and helps them automatically adapt and continue providing their service when adverse situations occur, e.g., node crashing and data discarding. However, due to the vulnerabilities in their implementation, blockchain systems may fail to recover from the error situations, resulting in permanent service disruptions. Such vulnerabilities are called resilience issues.

In this paper, we propose Phoenix, a system that helps detect and locate blockchain systems' resilience issues by context-sensitive chaos. First, we identify two typical types of resilience issues in blockchain systems: node unrecoverable and data unrecoverable. Then, we design three context-sensitive chaos strategies tailored to the blockchain feature. Additionally, we create a coordinator to effectively trigger resilience issues by scheduling these strategies. To better analyze them, we collect and sort all strategies into a pool and generate a reproducing sequence to locate and reproduce those issues. We evaluated Phoenix on 5 widely used commercial blockchain systems and detected 13 previous-unknown resilience issues. Besides, Phoenix successfully reproduces all of them, with 5.15 steps on average. The corresponding developers have fixed these issues. After that, the chaos resistance time of blockchains is improved by 143.9% on average. This indicates that Phoenix can significantly improve the resilience of these blockchains.

CCS CONCEPTS

• **Security and privacy → Vulnerability scanners**; **Distributed systems security**; • **Software and its engineering → Software testing and debugging**.

*Yuanliang Chen has contributed equally to this work.
†Yu Jiang is the corresponding author.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
CCS '23, November 26–30, 2023, Copenhagen, Denmark
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0050-7/23/11.
https://doi.org/10.1145/3576915.3623071

KEYWORDS

blockchain systems, chaos engineering, bug reproduce.

ACM Reference Format:
Fuchen Ma, Yuanliang Chen, Yuanhang Zhou, Jingxuan Sun, Zhuo Su, Yu Jiang, Jiaguang Sun, and Huizhong Li. 2023. Phoenix: Detect and Locate Resilience Issues in Blockchain via Context-Sensitive Chaos. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3623071

1 INTRODUCTION

Resilience is the ability of a blockchain to adjust so it can sustain its normal functioning in the face of changes and disturbances. Since blockchain systems always operate in a volatile environment, anomalies may cause severe consequences. For example, in public chains like Ethereum [21], anyone with a completely different environment can join the network. Due to this feature, failures such as node crashes and malicious attacks may occur frequently. Consortium chains like Hyperledger Fabric [38] tend to have a small group of nodes. Therefore, network fluctuations or hardware failures at any node can have a significant impact on the entire system.

To resist anomalies and provide resilience, blockchain systems design some coping mechanisms, such as consensus and synchronization. For example, when a node's data is dropped, it can synchronize from other nodes and continue to provide services. However, while implementing such mechanisms, some vulnerabilities may exist and inhibit the system from recovering from errors. Such vulnerabilities that cause service recovery failures are known as resilience issues. In particular, we identified two typical resilience issues in blockchain systems: **1) Node unrecoverable.** This indicates that the node is permanently lost from the group. For example, a node crashes due to a memory leak [12] or a program panic [13] and cannot be restarted automatically. Alternatively, a node may become separated from other nodes due to a hard fork [30], which results in node isolation. **2) Data unrecoverable.** This means that the consistency [4] or immutability [35] of the blockchain system is permanently broken. For example, a node whose data is inconsistent with other nodes cannot be synchronized to the correct result [72]. Alternatively, blockchain nodes may store invalid data [41]. Both

of these problems can have serious security consequences. Specifically, node unrecoverable issues [51] may lead to DDoS attacks and prevent blockchain systems from providing services. While data unrecoverable issues [24] can result in the acceptance of invalid transactions due to data corruption, such as double spending, and lead to huge asset losses.

Chaos testing is considered a useful method for assessing the resilience of distributed systems. For example, ChaosBlade [7] is an open-source fault injection tool following the principles of chaos testing. It has been widely used in cloud native systems. Apart from that, some chaos tools also target at blockchain systems, such as CHAOSETH [73]. Rather than a bug detecting tool, CHAOSETH monitors the performance variance of blockchains under chaos conditions. However, all existing chaos tools cannot effectively analyze resilience issues in blockchain systems through their testing strategies due to two main challenges.

The first challenge is that chaos strategies in existing chaos tools ignore the runtime context information and cannot effectively trigger resilience issues. Traditional chaos strategies can only modify the node's environment at a coarse-grained level. The context information of the blockchain runtime refers to key execution phases, such as reading/writing block data, sending/receiving consensus packets, requesting/responding to a data synchronization process, etc. These contexts are necessary for resilience issues. For example, a resilience issue may only be triggered when the node is isolated after a successful block synchronization. Moreover, strategies without knowing such context that make anomalies all the time may not satisfy the trigger conditions of this issue. Thus, it requires context-sensitive chaos strategies tailored to the characteristics of the blockchain to detect resilience issues.

The second challenge is that locating and reproducing a bug in distributed scenarios is hard. It is not enough to just detect a resilience issue. It is more important to steadily reproduce the bug to uncover why and how it happened. This is essential to help developers quickly remediate the issue and check whether the bugs are fixed. However, bug reproduction often requires accurate documentation of distributed behaviors and precise control over the replay of those behaviors. And existing chaos tools are lack of such reproducing processes. Resilience problems in blockchain systems are always triggered by a combination of behaviors. For example, a data loss behavior should be conducted first to trigger the synchronization mechanism. Then a data pollution needs to be performed during the synchronization process to trigger a data unrecoverable issue. It is challenging to generate a reproduction sequence to collect all these behaviors.

In this work, we design and implement Phoenix, a system to detect and locate the resilience issues in blockchain systems by performing context-sensitive chaos testing and bug reproducing. First, Phoenix designs 3 context-sensitive chaos strategies, including block/transaction data corruption, byzantine attack, and node partition. All of them are instrumented into the system's source code based on corresponding contexts. With the help of a test coordinator, Phoenix performs a test round along with a check round by sending different signals to the SUT (System Under Test). In the test round, the blockchain system is injected with various chaos strategies to mimic adverse situations. While in the check round, Phoenix checks whether the nodes and the data have recovered from the chaos. If a resilience issue is detected, Phoenix will generate a reproduction sequence by collecting and ordering all the chaos strategies from each distributed node. A reproduction coordinator sends reproduction signals to each node and precisely controls the replay of each strategy. This way, Phoenix can effectively detect and reproduce resilience issues in blockchain systems to help developers better understand errors or risks and react to them when they inevitably arrive.

We implement and evaluate the effectiveness of Phoenix on 5 commercial blockchain systems, including Hyperledger Fabric [38], FISCO-BCOS [25], Quorum [8], Go-Ethereum [21] and Binance Smart Chain [6]. Currently, Phoenix detected 13 previously unknown resilience issues in these systems. In addition, Phoenix successfully reproduces all of the found issues steadily by utilizing 5.15 steps in the reproduction sequence on average. The corresponding developers have fixed all the issues found. After that, the chaos resists time of all five blockchains can be extended by 143.9%. These statistics demonstrate that the resilience of blockchain systems can be significantly improved.

In general, we make the following contributions:

- We propose a system that performs chaos testing with context-sensitive strategies tailored to blockchain features. It also supports bug reproduction for resilience issues in blockchain systems.
- We design and implement Phoenix. With two coordinators, Phoenix controls resilience issues testing and reproducing processes by sending various signals.
- We evaluated Phoenix on 5 widely used blockchain systems. We will open-source Phoenix [1] for practical usage. For now, Phoenix has detected 13 resilience issues and successfully generated reproduction sequences for all of them.

2 BACKGROUND

2.1 Blockchain and its Resilience

Blockchain is a decentralized system which is maintained by a network of nodes. These nodes work together to validate and process the transactions and store the execution results. Blockchain systems can be divided into three types according to their requirements: private blockchains, public blockchains, and consortium blockchains.

A private blockchain is a permissioned system that is operated in a closed network. Such blockchain is mainly used by an enterprise or an organization for internal applications. The other two types of blockchain systems run in an open environment. A public blockchain is a permissionless distributed ledger that provides applications such as cryptocurrency exchanging [65] and decentralized finance [11]. Some well-known public blockchains include Ethereum [21], and Binance Smart Chain [6]. Unlike a public blockchain, a consortium blockchain using the underlying platform such as Hyperledger Fabric [38], FISCO-BCOS [25] or Quorum [8], is maintained by several organizations with exact business needs. Public and consortium blockchain systems run in a complex environment where node crashes, network fluctuations, and attacks may usually happen. To provide continuous service, blockchain

[1] Phoenix at: https://anonymous.4open.science/r/Phoenix-20FE/

systems must implement mechanisms to resist these situations and provide resilience. Generally, they provide the following two ways to recover from failures:

Consensus. Consensus is a necessary process to make sure that each transaction has a certain execution result under a decentralized system. Since node crashes and byzantine attacks are common in blockchain systems, developers equip the blockchain systems with both CFT (Crash Fault Tolerant) and BFT (Byzantine Fault Tolerant) based consensus algorithms such as Raft [55], PBFT [5], and PoS [28], and PoW [29]. These consensus protocols provide resilience by ensuring that the blockchain remains consistent though certain nodes are down or malicious.

Data Synchronization. Apart from the consensus protocols. Blockchain systems implement a synchronization process to resist the situations where nodes are isolated from the main network. When a partition occurs in the blockchain system, the data stored by each node may be different. At this time, the node will try to synchronize the latest block data of other nodes to update the local chain. In this way, the system provides resilience and achieves partition tolerance.

However, due to the fact that there are inevitable implementation vulnerabilities in these mechanisms, in some cases the blockchain may fail to provide resilience and lead to permanent service failure.

2.2 Chaos Testing

Chaos testing was first proposed in 2010 by the developers of Netflix [3] when they decided to migrate their infrastructure from physical machines to the cloud. The key insight of chaos testing is to evaluate the ability of distributed systems to cope with various contingencies in a complex cloud environment. To ensure that Netflix can provide a stable service, the developers created ChaosMonkey [52], which is considered the first chaos testing tool.

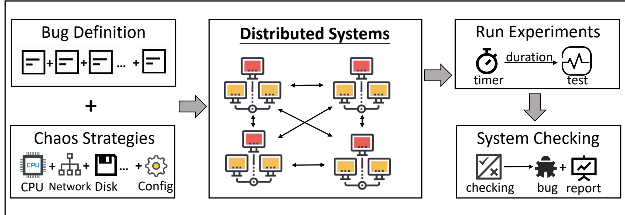

Figure 1: The general workflow of chaos testing on a distributed system. With predefined assertions and chaos strategies, the system runs the experiment for a duration. It then checks whether a bug exists and generates reports.

Basically, the general workflow of chaos testing is shown in Figure 1. There are 4 basic steps for chaos testing: ① Prepare some assertions of the SUT to define bugs for this testing. ② Design various chaos strategies for the SUT. In this step, the developer should summarize some adverse situations that could occur in the deployment environment and inject them into the SUT. Usually, the strategies may include CPU controlling, network delay, disk occupation, or system configuration modification.③ Run the chaos experiments. The system will be tested by predefined strategies within a duration. ④ The SUT will be checked by the assertions and gives out bug information and reports. If something wrong happens, the developers may know that there are some bugs in the system that may lead to service failure. Followed by these steps, chaos testing has successfully detected plenty of bugs in distributed systems before they are deployed for practical usage.

Originally, Netflix introduced ChaosMonkey for randomly terminating instances to simulate failures. This tool is useful for imitating crash failures in a cloud system. However, a weakness of ChaosMonkey is that the failure occurs randomly without the developers' control. To reduce randomness, Netflix later developed Simian Army [53], allowing finer control over chaotic objects and duration. Simian Army introduces an engineering approach to chaos testing. However, the dimensions and types of faults it supports are not rich. To conduct more systematic chaos experiments, ChaosBlade combines multi-level chaos strategies like network disruption and CPU control. This enables chaos testing from various perspectives. For a specific system, chaos testing requires designing system-specific strategies. For example, to perform chaos on a DBMS, the strategies may tailor to transaction management and SQL optimization. As for blockchain systems, all existing tools lack strategies related to the blockchain context. While Phoenix identifies these weak spots and tailors blockchain-specific strategies to the system context.

3 OVERVIEW

3.1 Definition of Blockchain Resilience Issues

Threat Model: Throughout this paper, we use the following threat model. First, we formally define a blockchain network as $\phi = \{m_1, m_2, T_{chaos}, T_{recover}\}$. Specifically, m_1 means the number of normal nodes which perform honestly in the network. m_2 presents the chaos nodes under the attacker's control. Attackers can conduct chaos strategies, including data corruption, byzantine attacks, node partition, etc. T_{chaos} indicates the maximum time attackers execute a chaos attack. And $T_{recover}$ indicates the time it should take for the blockchain to recover to the normal state. $T_{recover}$ is positive infinity if the blockchain is unrecoverable. The proportion of chaos nodes should satisfy the fault tolerance mechanisms of the SUT. For most public blockchain systems, $m_2/(m_1 + m_2)$ should be smaller than $1/2$. While for most consortium blockchain systems, it should be smaller than $1/3$.

For a blockchain system under test, after a chaos attack lasting $T_1 (T_1 \leq T_{chaos})$, given a finite time T_2 for recovering, the entire blockchain system should return to the normal state and be able to provide all functional services normally. Otherwise, the blockchain system is in an unrecoverable state, and we identify there is a resilience issue in the blockchain system. Specifically, we divide the blockchain unrecoverable state into two main types: Node Unrecoverable State and Data Unrecoverable State.

Formally, we consider a blockchain system under test with m_1 honest nodes taken from a finite set $\Pi = \{n_1, n_2, n_3, ...n_{m_1}\}$. We use the symbol Ψn_i presenting node n_i crashes down. Let the finite set $TX_i = \{tx_{i1}, tx_{i2}, tx_{i3}, ..., tx_{in}\}$ present the transaction pool of node n_i. A finite set $B_i = \{b_{i1}, b_{i2}, b_{i3}, ..., b_{ip}\}$ represents the local blockchain of node n_i, and each block $b_{ij} \in B_i$ contains a set of confirmed transactions. We also define the symbol $B_{polluted}$ as the set of block data polluted by attackers.

The Node Unrecoverable State includes two types of states in the blockchain systems. (1) Node crashes [64], where node stops

functioning and exits erroneously. Formally in LTL (linear temporal logic) [68], $\forall n_i \in \Pi, \Box(\neg \Psi n_i)$ — all nodes in blockchain systems should not crash down at any time. Otherwise, the node crashes state is triggered. (2) Transaction processing is stuck [67], where nodes stop handling transactions and can not be recovered automatically. In LTL, for each node, $n_i \in \Pi, \forall tx \in TX_i, \Diamond(tx \in B_i)$ — all transactions in blockchains should be processed and committed into a specific block eventually. Otherwise, the transaction process reaches a stuck state. If one of these two states occurs, then a node unrecoverable bug is found.

The Data Unrecoverable State also includes two types of states. (1) Data inconsistency [74], where data in distributed nodes become inconsistent and can not be recovered automatically. Formally in LTL, For each node, $n_i \in \Pi$, $\forall n_i, n_j \in \Pi$, $\forall k : 1 \le k$, $\Diamond(b_{ik} \equiv b_{jk})$ — all block data in different nodes should eventually keep consistent. Otherwise, a data inconsistency state is reached. (2) Polluted data storage [35], where the data polluted by attackers is committed and stored mistakenly. Formally in LTL, $\forall b_{polluted} \in B_{polluted}, \Box(b_{polluted} \notin B_i)$ — all polluted data should never be committed and stored in any block. Otherwise, a data polluted state is reached. If one of these two states occurs, a data unrecoverable bug is detected.

3.2 Motivating Example

Resilience issues are common in blockchain systems and can cause severe consequences. In this section, we give an example that describes a node unrecoverable issue in Go-Ethereum in 2023. The root cause of this vulnerability is discussed in the GitHub issue #26300 [37]. However, another 4 bugs [33, 36, 50, 61] are also caused by this vulnerability. This vulnerability typically occurs when a skeleton header reference is mistakenly deleted, which can cause the block synchronization process to get stuck and panic. Figure 2 shows the code where the vulnerability occurs.

Issue Analysis: As shown in the code snippet, the bug is triggered during the beacon chain synchronization process. A beacon chain [56] along with a skeleton chain [58] work together to enable the PoS consensus in Ethereum. During a data synchronization process, a node receives new blocks from the beacon chain. On the other hand, the skeleton chain is a cache that stores a contiguous header chain. During the synchronization, a skeleton chain tracks a series of potentially dangling headers until they are written into the local database.

This issue is triggered in a scenario described in Figure 3 and Figure 4. In the first phase shown in Figure 3, one Ethereum node successfully synchronized the block from the beacon chain with a height of 325. The results of this process are a single skeleton header (block N) and a subchain with a single block (head N, tail N) being stored on disk. Here, N is 325@0946fa, where 325 is the block height and the '0946fa' is the hash value of the imported number. After that, anomalies such as network disconnection occur to this node. As a result, this node is temporarily isolated (node partition) from the blockchain cluster.

In the second phase shown in Figure 4, the node recovers from the anomalies and reenters into the blockchain cluster. In this case, some new blocks in the beacon chain need to be synchronized to the local skeleton chain. To achieve this, it first calls the function 'processResponse' in line 1 in Figure 2. Currently, there are

```
1  func (s *skeleton) processResponse(res *headerResponse)
       (linked bool, merged bool) {
2    // only two subchains and the older one has only 1 block
3    case subchains == 2 && Subchains[1].Head==Subchains[1].Tail:
4  - log.Debug("...","head",Subchains[1].Head)
5  - rawdb.DeleteSkeletonHeader(batch, Subchains[1].Head)
6  + if Subchains[1].Head < Subchains[0].Tail {
7  +    log.Debug("...","head",Subchains[1].Head)
8  +    rawdb.DeleteSkeletonHeader(batch, Subchains[1].Head)
9  + }
10   ...
11 }
12 func (d *Downloader) findBeaconAncestor() (uint64, error) {
13   var linked bool
14   switch d.getMode() {
15   case FullSync:
16      //<--- SIGSEGV occurs when beaconTail is nil --->
17      linked = d.blockchain.HasBlock(beaconTail.ParentHash,
18                beaconTail.Number.Uint64()-1)
19   case SnapSync:
20      linked = d.blockchain.HasFastBlock(beaconTail.ParentHash,
21                beaconTail.Number.Uint64()-1)
22   ...
23 }
```

Figure 2: A node unrecoverable issue found in Go-Ethereum. Line 17 tries to access a nil beaconTail and trigger a SEGV signal. The root cause is due to line 4 in the function 'processResponse', which deletes a skeleton header mistakenly.

2 subchains now (the new beacon chain and the local skeleton chain), and the second subchain's head and tail have the same value (325@0946fa), which satisfies the case in line 3. Thus, as shown in line 5, the reference of the skeleton header is deleted. Afterward, as lines 16-18 show, when the node tries to find the beacon ancestor by fetching the beacon tail, it tries to read the deleted reference and triggers a SEGV signal. This bug has already been fixed [62] by adding an if branch, as shown in lines 6-9. In the patch, the node will only delete the reference when the head of the local skeleton chain is less than the tail of the beacon chain. This represents that the synchronization will start at the tail of the beacon chain and will never use the reference whose header is less than it. Thus, the deletion will always be safe.

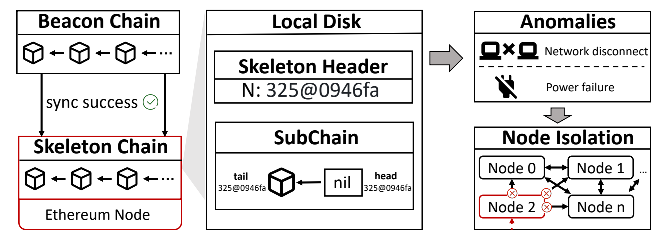

Figure 3: The first phase to trigger this bug. One node synchronized the block successfully from the beacon chain, and stored a header 325@0946fa and a subchain with only one block. Then due to anomalies, it is isolated from the cluster.

Challenges for analyzing such issues. This issue significantly affects the safety and availability of Ethereum. Unfortunately, it

is hard to analyze this issue in an effective way. As we illustrated before, there are two main challenges.

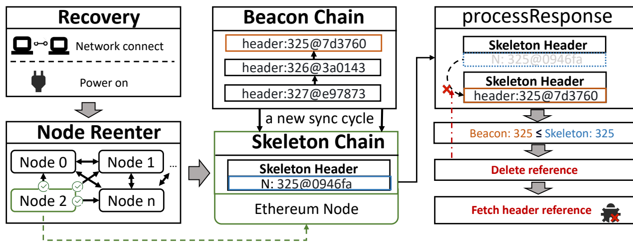

Figure 4: The second phase to trigger this bug. After the recovery, the node reenters the cluster and starts a new sync from the beacon chain. The node mistakenly deletes the skeleton reference which leads to the bug.

For the first one, it is hard for existing chaos tools to find this resilience issue due to their strategies lack of context information. Existing tools like ChaosBlade [7] support strategies such as CPU controlling and network delay in the system environment. In this situation, isolating a node with traditional strategies may prevent the trigger condition of the bug. For example, ChaosBlade can disconnect a node through traffic control and isolate the node all the time. However, during the anomalies, this node may not have stored a subchain containing only one block to disk, as it may not have just finished a successful sync. The condition in line 3 in figure 2 could not be satisfied as the current subchain's tail and head may not be the same. Thus, the bug will never be triggered. To detect such issues, context-sensitive chaos strategies need to be instrumented. In this case, node isolation strategies should be injected into the code where synchronization is successfully finished. That is why we design Phoenix with context-sensitive strategies and a coordinator to schedule them. Phoenix can detect the bug for this case by triggering a node partition strategy after a successful synchronization. Thus, the partition will happen precisely after the successful sync from the beacon chain. And after the partition finishes, the node will try to reenter the cluster and sync from the beacon chain. This will trigger the situation shown in Figure 4 and expose the bug.

For the second one, it is hard to find the resilience issue's root cause in a distributed environment. In this example, the first bug issue [36] was reported on September 16, 2022, the root cause [37] is described on December 04, 2022, and patch [62] is provided on January 09, 2023. It takes a long time to thoroughly understand the bug and find the proper fix. This is because the bug cannot be reproduced steadily. Thus, the developers may only know that a null pointer bug exists but do not know why it happens. Current chaos tools lack a bug reproducer to help developers understand what strategies trigger the detected bugs and reproduce them steadily. In this case, if the chaos tool can tell the developers that a node partition strategy after a successful synchronization process can reproduce this bug, the developers may understand the bug in a shorter time. That is why we equipped Phoenix with a resilience issue reproducer.

4 PHOENIX DESIGN

Figure 5 shows the overall design of Phoenix. The workflow of Phoenix is divided into two processes: the context-sensitive detecting process and the context-sensitive locating process. A context is defined as a pair: $< C, S >$, where C and S refer to consensus and data sync hooking positions. Specifically, for Phoenix on FISCO-BCOS, C contains before/after_preprepare, before/after_prepare, before/after_recover_response, before/after_viewchange and before/after_checkpoint. S includes before/after_block_request, before/after_tx_request, before/after_block_sync and before/after_sync_status. Context is generated by 3 steps: 1) Phoenix marks all the interesting positions in the source code. 2) Phoenix sets the pair by filling in the positions. 3) According to the pair, Phoenix adds corresponding chaos strategies.

Phoenix first defines three context-sensitive strategies in the context-sensitive detecting process, including block/transaction data corruption, byzantine attacks, and node partition. There are two phases in this process. The first phase is the chaos resisting phase, where the SUT keeps on running and tries to resist various chaos strategies performed by Phoenix for T_1 duration. While the second phase is the system recovery phase, where Phoenix stops all the chaos strategies and checks whether the SUT has recovered from the chaos within T_2 duration. The test coordinator executes these two phases in rotation. The resisting duration T_1 and the recovery duration T_2 of the SUT are controlled by two timers. The test coordinator communicates with the SUT through a signal channel. Phoenix then checks whether the system runs normally by two resilience issue checkers: a node checker and a data checker, corresponding to the two resilience issues we proposed.

In the context-sensitive locating process, Phoenix first collects all the performed chaos strategies from all the chaos nodes into a strategy pool and sorts them based on their timestamps. Then, Phoenix selects the latest 2^n strategies in the sequence. Here, n represents the reproduction rounds. It gradually increases from 1 until the bug is successfully reproduced. If 2^n exceeds the number of strategies in the strategy pool, then the reproduction fails. Like the chaos testing process, Phoenix uses a probability controller to schedule the reproduction steps for each chaos node by sending various signals. Phoenix checks whether the issue has been successfully reproduced with two reproduction result checkers. They compare the system states and logs when the issue occurs with the current system states and logs. In this way, Phoenix tries to detect and reproduce resilience issues effectively.

4.1 Context-Sensitive Detecting

In this section, we will explain how Phoenix performs context-sensitive testing for blockchain systems. First, we describe the context-sensitive strategies in Phoenix, then we introduce how the test coordinator works, and finally, we give the details of two resilience issue checkers.

4.1.1 Context-Sensitive Strategies. Phoenix defines three context-sensitive strategies specific to blockchain systems for chaos testing. As we described before, traditional chaos strategies such as CPU controlling or disk filling cannot effectively detect resilience issues in blockchain systems because they lack runtime context information. Based on the features of blockchain systems, Phoenix defines

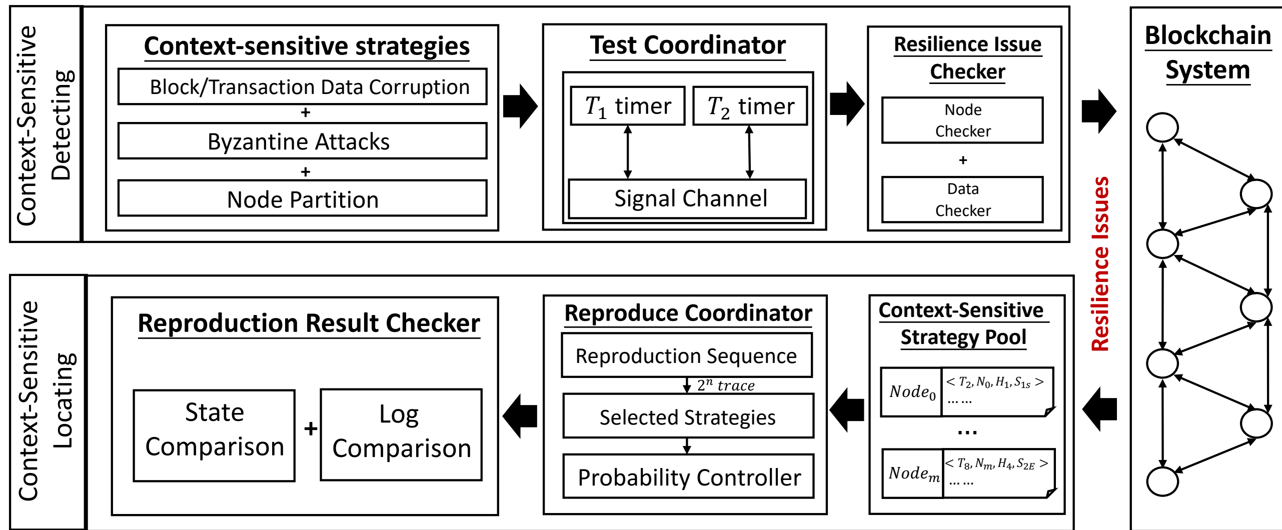

Figure 5: An overview of Phoenix design. There are basically two phases in Phoenix. (1) During the context-sensitive detecting phase, Phoenix injected three context-sensitive strategies into the SUT. The test coordinator sets two timers to control the testing time and checking time by sending different signals to the system nodes. Meanwhile, Phoenix uses two checkers to detect whether the SUT has resilience issues. (2) During the context-sensitive locating phase, Phoenix first collects context-sensitive strategies into a pool. Afterward, Phoenix uses a reproduce coordinator to generate and execute the reproduction sequence. It then checks whether the issue has been successfully reproduced by comparing the system's states and logs.

three specific strategies: block/transaction data corruption, byzantine attacks, and node partition.

Block/transaction data corruption. This strategy means deleting or polluting a blockchain node's block and transaction data. It is used to mimic the situation where a node's storage is faulty or maliciously altered. This is common in a blockchain system. Many accidents can lead to this situation, such as flaws in the underlying database software or physical storage device failures. To trigger resilience issues under this situation more effectively, Phoenix injects this strategy with a certain probability under two contexts: (1) a node needs to read the stored block data, and (2) it writes new data to the disk. Phoenix implements this strategy by removing the data files or appending dirty data to them.

Byzantine attacks. This strategy imitates byzantine attacks when hackers compromise a node. By adding cheating information in the consensus process, a hacker may gain profit by convincing other nodes to accept invalid execution results. Considering the severe consequences caused by such attacks, most of the consortium blockchains have supported BFT-based (stands for byzantine fault tolerant) consensus algorithms to achieve consistency (e.g., SmartBFT [22] in Hyperleger Fabric, QBFT [43] in Quorum and PBFT [5] in FISCO-BCOS.). Meanwhile, public blockchains leverage PoW [29] or PoS [28] consensus algorithms that resist byzantine attacks. However, implementing such algorithms may contain vulnerabilities that become resilience issues. Phoenix uses this strategy in two contexts: (1) before sending consensus packets and (2) after receiving consensus packets. To implement the byzantine attacks, Phoenix messes with the order of the sending packets, repeating some packets, modifying packet contents, and sending different packets to different target nodes. By doing so, nodes in the SUT may behave erroneously after receiving incorrect packets. Phoenix uses this strategy to test whether the SUT can resist such attacks.

Node Partition. This strategy simulates the situation where nodes are separated from others in a group. Performing this strategy will create partitions in the blockchain systems. For example, nodes shut down due to a power failure or a network connection problem. Phoenix instruments this strategy with a certain probability in two contexts: (1) after a synchronization process is finished successfully (the same as the motivating example.), (2) before sending new synchronization requests. To achieve this, Phoenix first disconnects the chaos nodes' network and then reconnects it.

4.1.2 Test Coordinator. Phoenix contains a test coordinator to schedule the execution of chaos strategies. The main function of this coordinator is to let the SUT know when chaos tests should be performed and when checks should be performed. To achieve this goal, the coordinator implements a phase controller containing two timers and a signal channel.

In the chaos testing phase, whose duration is set by the T_1 timer, the SUT performs chaos strategies defined by Phoenix. While in the recovery phase, whose duration is set by the T_2 timer, Phoenix stops all the strategies and checks whether there are resilience issues in the SUT. Phoenix tells the SUT the current phase by sending two signals: *SIGCHAOS* and *SIGRECOVER*. When a node receives *SIGCHAOS*, it executes the predefined chaos strategies randomly. In contrast, if a node receives *SIGRECOVER*, it stops all the running strategies and does not perform new chaos strategies.

The workflow of the test coordinator during the chaos testing phase can be referred to as Algorithm 1. The algorithm inputs contain the chaos testing duration T_1 and the system recovery duration T_2. The program contains a *while true* loop, as shown in

Algorithm 1: Chaos phase controlling

```
Input  : Chaos testing duration: T1, System recovery
         duration: T2
1  Function chaosControl(T1, T2):
2      setCurrentPhase (-1);
3      while true do
4          start = clock();
5          // In the chaos testing phase
6          while clock() - start < T1 do
7              if getCurrentPhase() != 0 then
8                  setCurrentPhase(0);
9                  sendSignal(SIGCHAOS);
10             end
11         end
12         // In the recovery phase
13         while clock() - start < T1 + T2 do
14             if getCurrentPhase() != 1 then
15                 setCurrentPhase(1);
16                 sendSignal(SIGRECOVER);
17                 stopAllStrategies();
18             end
19             suc = performChecking();
20             // SUT is recovered
21             if suc == true then
22                 actual_T2 = clock() - start - T1;
23                 record(actual_T2);
24                 break;
25             end
26         end
27         // SUT is in unrecoverable state
28         if clock() - start > T1 + T2 then
29             reportIssue();
30         end
31     end
32 End Function
```

line 3 in the algorithm. In the loop, the coordinator first gets the current timestamp by using the function *clock()* as shown in line 4. Lines 6-11 describe the chaos testing phase. Phoenix first gets the current phase from a global variable (0 indicates the chaos testing phase, and 1 indicates the recovery phase.). If the variable is not 0, Phoenix updates it to 0 and sends the signal *SIGCHAOS* to chaos nodes. The recovery phase is shown in lines 13-26. Line 14-18 shows a similar process as the last phase, which resets the global variable to 1 and sends the *SIGRECOVER* signal. In addition, the coordinator also stops all the running chaos strategies in this phase, as line 17 shows. After that, the coordinator checks resilience issues to identify whether the SUT recovers to the normal state. If the SUT is recovered, the $performChecking()$ returns true. Otherwise, it returns false. As lines 21-25 show, if the result is true, then Phoenix calculates and records the actual recovery time of the system and jumps out of the checking phase. If the result is false, Phoenix performs the checking repeatedly until the duration T_2 is up. As lines 28-30 show, if *suc* is still false after T_2, the blockchain system is unrecoverable. And Phoenix considers there is a resilience issue.

The coordinator generates strategies in the following manner:

- **Distribution**: The distribution of strategies in a sequence aligns with the actual behaviors of blockchains. More frequent behaviors are associated with a higher distribution of corresponding strategies. For instance, consensus processes occur more frequently than data sync, resulting in consensus-based strategies having broader distributions than data sync-based strategies.
- **Heuristics**: The heuristics employed by Phoenix prioritize more frequent behaviors as being more significant and susceptible to resilience issues. Thus we instrument the chaos strategies in these frequent behaviors, like conducting byzantine attacks before sending consensus packets.

4.1.3 Resilience Issue Checker. The resilience issue checker is responsible for checking whether the SUT recovers to the normal state after stopping the chaos strategies within T_2. There are two checkers in Phoenix's design, corresponding to the two types of resilience issues respectively. The first checker is the *Node Checker*. It checks whether each node in the SUT is still alive and works functionally. To be specific, the node checker first checks whether the process of each node still exists. If the process of each node is alive, the checker will then send a series of transactions to each node and see whether they can be processed correctly. In this way, the checker covers both node crashes and transaction processing stuck introduced in the node unrecoverable issue.

The other checker is the *Data Checker*. It checks whether the data in each node are still consistent and valid. Specifically, the checkers first check whether each node's block is equivalent. Secondly, the checker checks whether any nodes commit and store the polluted data generated by the chaos testing. This checker covers the data unrecoverable issues, including data inconsistency and polluted data storage, as described in Section 3.

4.2 Context-Sensitive Locating

In this section, we introduce how Phoenix locates and reproduces the detected resilience issues. First, we introduce how to collect the context-sensitive strategy pool, then we describe how the reproduction coordinator works, and finally, we give the details of two reproduction result checkers.

4.2.1 Context-Sensitive Strategy Pool. The reproduction process first maintains a strategy pool consisting of all the performed chaos strategies from all chaos nodes in the SUT. As Figure 5 shows, a strategy is represented as a quaternion: $< T_i, N_j, H_m, S_{nS}/S_{nE} >$. T_i represents the timestamp of the strategy. N_j means the id of the node that performed this strategy. H_m refers to the hook context where the strategy is injected. In the implementation of Phoenix, the context is represented as a variable with *enumerate* type. The last element S_{nS} or S_{nE} indicates the strategy id and the state of the strategy (S means start while E means end.). For example, $< T_4, N_0, H_3, S_{1S} >$ describes a strategy with the id 1 has started at the time T_4. This strategy is performed by the node N_0 at the context H_3. (H_3 represents the fourth element in the enumerate.) While $< T_5, N_0, H_3, S_{1E} >$ means that this strategy has ended at T_5.

During the chaos testing process, each chaos node in the SUT records the strategies performed by itself. These strategies are stored by timestamps in various files named after each node's id. After detecting a resilience issue, Phoenix collects the strategies from each node's file into the pool. Currently, the pool contains all the strategies performed by each chaos node. Phoenix then sorts all the strategies according to their timestamps. In this way, the first strategy in the pool is the oldest, while the last strategy is the newest one. As shown in Figure 6, $< T_{25}, N_3, H_1, S_{4E} >$ is the newest strategy which happens at T_{25}.

4.2.2 Reproduce Coordinator. The reproduce coordinator is responsible for guiding the SUT to perform precisely the same strategies as those that have triggered the resilience issue in the reproduction process. Figure 6 shows the workflow of this coordinator in the reproduction process.

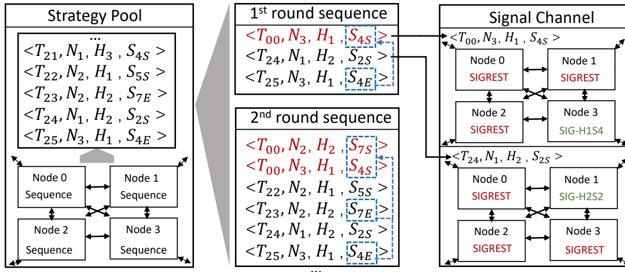

Figure 6: The reproduction workflow of Phoenix. It first selects the reproduction sequence for each round and then sends signals to different nodes to guide their behaviors.

The coordinator first selects 2^n latest strategies from the strategy pool. Here n represents the number of the current reproduction round. In the 1st round, the Phoenix coordinator selects the latest 2 strategies in the pool, which are $< T_{24}, N_1, H_2, S_{2S} >$ and $< T_{25}, N_3, H_1, S_{4E} >$ in Figure 6. However, the latest strategy describes an ending strategy for which we could not find a corresponding starting strategy in the current sequence. In this situation, Phoenix appends an initial strategy to the sequence, as the red strategy shows in the figure. For the first round sequence, the appended strategy is $< T_{00}, N_3, H_1, S_{4S} >$. T_{00} here is a special timestamp symbol which means 'before all other strategies'. As we can see in the 2nd round sequence in the figure, Phoenix has to append two starting strategies with T_{00}. In this situation, the two strategies will be executed simultaneously if they are related to different nodes. If two strategies with T_{00} point to the same node, they will be executed in the order of their corresponding ending strategies.

After selecting the proper strategy sequence for each reproduction round, Phoenix executes them sequentially by sending signals to each SUT node. For example, in the 1st reproduction round, Phoenix guides the SUT to execute the first strategy: $< T_{00}, N_3, H_1, S_{4S} >$. This strategy is performed by node 3 at position H_1, and the strategy id is 4. Thus, Phoenix will send a signal *SIGREST* to all other nodes and send a signal *SIG-H1S4* to node 3. The signal *SIGREST* tells a node that it should stop performing any chaos strategy currently. To achieve this, each node will just adjust its chaos execution probability to 0% by receiving this signal. Thus, all nodes except for node 3 will stop performing any chaos strategy. While the signal *SIG-H1S4* tells a node to execute strategy with the id S_4 at the position H_1. Similarly, the corresponding node adjusts its chaos execution probability at H_1 to 100%. And node 3 will certainly execute the strategy S_4. After this strategy is executed successfully, a signal *SIGDONE* is sent to the coordinator from node 3, and Phoenix switches to the next strategy in the sequence, which is $< T_{24}, N_1, H_2, S_{2S} >$. It sends *SIGREST* to node 0, node 2, and node 3 and sends *SIG-H2S2* to node 1. Thus, node 0, node 2, and node 3 will not execute any chaos strategy, and node 1 will certainly execute strategy S_2 at position H_2 In this way, Phoenix can guide the SUT to perform specific strategies in the reproduction process.

4.2.3 Reproduction Result Checker. After performing all the strategies in the sequence of the current round, Phoenix first leverages the resilience issue checker, as we introduced in Section 4.1.3 to check whether a bug occurs. If not, it indicates that the current reproduction round has failed. Phoenix tries to reproduce the bug with more strategies in the next round until all of the strategies in the pool are consumed. If there is a bug in the current reproduction round, Phoenix will use two reproduction result checkers to check whether the bug is the same as the found issue. The first checker is a state comparison checker. It checks if the current bug has the same unrecoverable state as the original one. The unrecoverable state of a resilience issue means the consequence it causes, such as node crashes or data inconsistency. If they have the same state, Phoenix will then use the second checker, a log comparison checker. This checker compares the current log information of each node to tell whether the two bugs are the same. This checker only focuses on some important information about the current issue. For example, if the issue leads to a node crash, this checker will focus on the call stack that triggers the crash. As for data inconsistent issues, it checks the execution result in the log to judge whether they are inconsistent in the same way. If any checkers fail, Phoenix considers reproduction unsuccessful in this round. It moves to the next reproduction round and selects more strategies from the pool.

5 IMPLEMENTATION

In this section, we describe some implementation details of Phoenix. We implemented and evaluated Phoenix in 5 blockchain systems, Hyperledger Fabric (version 2.3.0 [39]), FISCO-BCOS (version 3.1.0 [27]), Quorum (version 1.1.0 [10]), Go-Ethereum (version 1.10.25 [57]) and BSC (version 1.1.17 [2]). They are chosen because of their popularity and diversity. Hyperledger Fabric is one of the most popular enterprise-grade blockchains. It has been widely used in many industrial environments, such as A.P. Moller-Maersk, Allianz, Ant Group, Tencent, etc. FISCO-BCOS is another popular financial-grade consortium blockchain that has already been applied in many financial areas, e.g., loans. Quorum is a consortium blockchain protocol forked from the well-known Ethereum blockchain protocol [32]. While Go-Ethereum and BSC (Binance Smart Chain) are well-known and most active public chains that support dapps in the world. All 5 blockchain systems come from different organizations. Fabric is developed by IBM in Go, FISCO-BCOS by WeBank in C++, Quorum by ConsenSys in Go, Go-Ethereum by Ethereum Org in Go, and BSC by Binance in Go. Implementation and evaluation

of these blockchain systems can demonstrate that Phoenix is a cross-platform testing framework with high scalability.

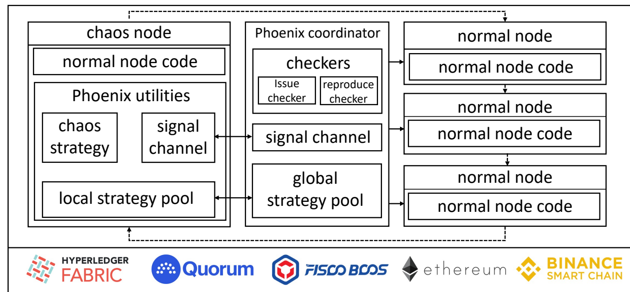

Figure 7: The implementation architecture of Phoenix. Phoenix utilities are packaged as a library and inserted into the node code. While Phoenix coordinator is a binary program that runs independently from the nodes in the SUT.

Figure 7 shows the implementation architecture of Phoenix. The key components of Phoenix are divided into two parts: Phoenix utilities and Phoenix coordinator. Phoenix utilities are packaged into a library. The blockchain node's original code will link it during the compilation. The utilities include predefined chaos strategies, a signal channel that guides the node to receive and send signals to the Phoenix coordinator, and a local strategy pool that records the strategies executed by the current chaos node. While the other components of Phoenix are compiled into a binary program named Phoenix coordinator. This program contains two checkers, a signal channel, and a global strategy pool that collects all the performed strategies from each chaos node's local strategy pool. The signal channel in the utilities is responsible for sending *SIGDONE* and receiving all other signals. While the signal channel in the coordinator sends *SIGCHAOS*, *SIGRECOVER*, etc, and receives *SIGDONE*.

Phoenix Deployment. During the deployment, Phoenix utilities are attached to 1/3 or 1/2 of the nodes in the SUT. This is to comply with the limits of related consensus protocols. The workload of a blockchain generally consists of various transactions based on smart contracts. To generate the workloads for each blockchain, we draw on each system's transaction generation program. Specifically, for Hyperledger Fabric, we use the smart contracts provided in the samples [40] given by the developers. For FISCO-BCOS, we use its stress testing scripts [26], which generate plenty of transactions based on a bank contract. For Quorum, Go-Ethereum, and BSC, we utilize a transaction firing tool called chainhammer [19]. To deploy Phoenix, one should start a group of blockchain nodes, several of which are attached to Phoenix utilities. Afterward, start the Phoenix coordinator and workload generation programs.

Phoenix Adaption. In the Implementation and Evaluation sections, we only adapted Phoenix to 5 commonly-used blockchain systems: Hyperledger Fabric, FISCO-BCOS, Quorum, Go-Ethereum, and BSC. They are widely used and implemented in different ways, demonstrating that Phoenix effectively detects resilience issues. However, Phoenix can be easily adapted to a new blockchain system by the following three steps:

(1) Locate the source codes which conduct the consensus and data sync, like message-passing and storage interfaces. This step marks the exact positions that chaos strategies should be injected. According to the context information, different positions are marked with flags to indicate which strategy should be injected. For example, the code position before sending a consensus packet will be marked as 'Byzantine attacks,' which means a byzantine chaos strategy should be added here.
(2) Inject the chaos strategies implemented in the Phoenix utility library. This injects proper strategies as well as the signal channels into the marked positions. Each strategy is equipped with a signal channel that sends and receives corresponding signals, which tells Phoenix whether the current strategy is triggered.
(3) Add issue checkers to monitor node processes and logs. Both resilience issue checker and issue reproduction checker should be tailored to the concrete log files of the target blockchain system. Specifically, for the resilience issue checker, developers should check the node liveness and functionality as well as the data inconsistency and pollution by identifying the log information of each node. Similarly, developers also need to implement a concrete way of comparing the log and states of the bugs during the reproduction process.

In addition, developers may use a different way to generate the workload in another blockchain.

6 EVALUATION

To evaluate the effectiveness of Phoenix, we compared it with state-of-the-art chaos tools: ChaosBlade [7] and Jepsen [42]. In addition, we also compared Phoenix with chaos tools targeted at blockchain systems like CHAOSETH [73] and BlockBench [15]. Considering that they do not aim at detecting bugs, we enhanced them with Phoenix's Resilience Issue Checkers for a fair comparison. Another tool named Hermes [49] detects bugs in BFT protocols by corrupting packet content. We also compare Phoenix with it in our evaluation. We ran a blockchain network of 10 nodes locally. The binary of FISCO-BCOS is hardened by AddressSanitizer [16] to detect latent bugs. For Phoenix and all other tools, we set up a group with 10 nodes for each target blockchain system and set 3 of them as nodes conducting chaos strategies. All the experiments are conducted several times on a 64-bit machine with 128 cores (AMD EPYC 7742). The OS of the machine is Ubuntu 20.04.1 LTS, and the main memory is 512 GB. We design experiments to address these research questions:

- **RQ1:** Is Phoenix effectively finding resilience issues in real-world blockchain systems?
- **RQ2:** Can Phoenix cover more blockchain systems branches than other tools?
- **RQ3:** Does Phoenix reproduce resilience issues accurately and quickly?
- **RQ4:** Can Phoenix efficiently help improve the resilience of blockchain systems?

6.1 Resilience Issues Detection

We ran Phoenix on all 5 blockchain systems. Since nodes in Fabric run in docker containers and tools, including ChaosBlade and Jepsen do not support it, we do not run them on Fabric. While CHAOSETH only supports Ethereum, we ran it only on Quorum, Go-Ethereum, and BSC (Quorum and BSC are both based on the

Table 1: Previously-unknown Resilience vulnerabilities found by Phoenix in 24 hours on 5 commonly-used blockchain systems. Phoenix detected 3, 4, 3, 2 and 1 bugs in Fabric, Quorum, FISCO-BCOS, Go-Ethereum and BSC respectively.

#	Platform	Bug Type	Bug Description	Identifier
1	Fabric	Node Unrecoverable	SIGSEGV: Node crashed down when starting syncing data after data chaos.	Bug#3879
2	Fabric	Node Unrecoverable	Node panic in runtime: nil pointer dereference in ledger snapshot comparison.	Bug#3878
3	Fabric	Data Unrecoverable	Constantly refusing blocks from the network when syncing block data.	Bug#3872
4	Quorum	Data Unrecoverable	After block data chaos, Downloader in syncing process continuously failed.	Bug#1589
5	Quorum	Node Unrecoverable	Node crashes down after receiving a polluted block in the block syncing process.	Bug#1588
6	Quorum	Node Unrecoverable	Node breaks down due to the data race in graphql, missing atomic operation.	Bug#1638
7	Quorum	Data Unrecoverable	Node stop syncing block with others after mistakenly executing data rollback.	Bug#1651
8	FISCO-BCOS	Data Unrecoverable	The execution results of transactions are not correct. Account balance is wrong.	Bug#3177
9	FISCO-BCOS	Node Unrecoverable	SIGSEGV: Node crashed when executing multiple transactions during the chaos.	Bug#3271
10	FISCO-BCOS	Node Unrecoverable	Nodes are stuck and stop handling transactions due to header polluted in block sync.	Bug#3307
11	Go-Ethereum	Data Unrecoverable	Geth client constantly fails to handle tx due to data race after plenty of data drops.	Bug#27480
12	Go-Ethereum	Node Unrecoverable	Node crashes down due to nil pointer dereference error when sync re-starts.	Bug#27173
13	BSC	Data Unrecoverable	Node's state cannot update after multiple snapshot pending processes.	Bug#1593

Table 2: Bugs found by Phoenix and other chaos tools. Other tools detect less than 3 bugs due to chaos strategies without context. While Phoenix detects 13 unknown resilience issues.

Tool Name	Bugs Found	Bugs Number
Phoenix	13	bug#1-bug#13
ChaosBlade	3	bug#6, bug#8, bug#9
Jepsen	3	bug#6, bug#8, bug#9
CHAOSETH	1	bug#6
BlockBench	1	bug#6
Hermes	1	bug#9

code of Ethereum). And for Hermes, we ran it on Fabric, Quorum, and FISCO-BCOS, which support BFT algorithms. Phoenix detects 13 previously unknown resilience vulnerabilities, including 3 in Fabric, 4 in Quorum, 3 in FISCO-BCOS, 2 in Go-Ethereum, and 1 in BSC. Their details are listed in Table 1. On average, Phoenix needs 31 samples and costs 24.6 minutes to trigger a bug. The reason why public blockchains have relatively fewer resilience issues is that they are constantly exposed to chaotic environments and have already unwittingly been subject to some common anomalies.

The corresponding vendors confirmed and repaired all the bugs at the time of paper submission. 7 bugs (#1, #2, #5, #6, #9, #10, #12) are of the type 'Node unrecoverable.' They cause the nodes in the blockchain network to crash down or transactions to get stuck and can not be recovered. Eventually, the blockchain system is out of service. These bugs can lead to DDoS attacks. 6 bugs (#3, #4, #7, #8, #11, #13) are of the type 'data unrecoverable.' This type of bug makes the block data of each node in the blockchain network become inconsistent or invalid and can not be recovered, leading to data loss. These bugs may lead to the acceptance of an incorrect transaction result and cause huge asset losses. For example, bug#8 leads to the incorrect calculation results of the user balance.

To analyze the false positives, we manually checked all the issues detected by Phoenix, and no false positives were found. This is because the maximum recovery duration T_2 we set in the experiments is 10 minutes.According to our recovery experiment in Section 6.4, the maximum recovery is less than 5 minutes. Hence, 10 minutes is long enough to eliminate the false positives.

The bug detection results of other tools are listed in Table 2. In our experiments, ChaosBlade and Jepsen only detected 3 resilience bugs (bug #6, bug #8, and bug #9). However, they did not find the rest of the 10 bugs because these resilience bugs are hidden in the relatively deep logic of the blockchain system. To trigger them, context-sensitive chaos strategies should be performed. As for CHAOSETH and BlockBench, they only detect 1 issue (bug#6). They cannot detect the rest resilience issues because their strategies are plain and lack context information. They only inject errors in system call invocation and ignore blockchain features like data synchronization and consensus. For Hermes, it only detects 1 issue (bug#9) because it lacks strategies such as data polluting, network partitions, and sending different packets to different nodes. With the help of the well-designed chaos strategies, Phoenix successfully detected all 13 resilience bugs, proving the effectiveness of Phoenix in detecting resilience issues in real-world blockchain systems, which adequately answers **RQ1**. Compared with other tools, Phoenix found all the bugs that they found.

6.1.1 Case Study. Now we use one case to illustrate how the resilience issues detected by Phoenix affect the whole blockchain system. **This case is the bug #1 listed in Table 1**. This bug is a Node Unrecoverable bug that causes some nodes to crash and can not be recovered in the blockchain network. It is found in version 2.3 of Fabric. The code snippet in figure 8 describes the detailed information of this resilience vulnerability.

Function 'Release()' is implemented to disconnect the node from the database after the block data is updated. In line 14, the $i.db$ will be set to nil. In line 13, $i.db$ will be accessed when executing $atomic.AddInt32()$. When the function 'Release()' is called in high concurrency, invalid memory panic occurs when $i.db$ is set to nil by other threads before executing code in line 16. As a result, the nodes crash, stop all their functional services, and cannot recover automatically. In our experiments, this Node Unrecoverable bug was only found by Phoenix. In Phoenix's chaotic situations, many malicious database access requests are sent to normal nodes. After receiving them, nodes have to access their local data frequently in

```
func (i *dbIter) Release() {
     if i.dir != dirReleased {
         runtime.SetFinalizer(i, nil)
       if i.releaser != nil {
         i.releaser.Release()
         i.releaser = nil
       }
       i.dir = dirReleased
       i.key = nil
       i.value = nil
       i.iter.Release()
       i.iter = nil
       atomic.AddInt32(&i.db.aliveIters, -1)
       i.db = nil
   }
}
```

Figure 8: A Node Unrecoverable bug that can crash normal nodes in the fabric SmartBFT blockchain network.

a short time. Thus, there is a high probability that they will execute the code in parallel and eventually trigger the bug.

6.2 Testing Coverage

In this section, we calculated the branch coverage of each blockchain system under Phoenix and other state-of-the-art tools. To achieve coverage, we use different ways for different systems. For Fabric, Quorum, Ethereum, and BSC, which are written in the Go language, we use gtest [18] to fetch the branch coverage. As for FISCO-BCOS, we use gcov [17] to collect the coverage information. The results are shown in Table 3.

Table 3: The branch coverage of Phoenix and other tools on 5 blockchain systems in 24 hours. '-' means the tool does not support the corrsponding system.

	Phoenix	ChaosBlade	ChaosETH	BlockBench	Jepsen	Hermes
Fabric	**13,126**	-	-	10,588	-	11,983
Quorum	**11,932**	9,903	9,263	9,067	10,263	9,816
FISCO-BCOS	**31,463**	24,202	-	-	21,057	23,180
Ethereum	**11,663**	9,682	9,141	8,932	9,527	-
BSC	**12,058**	9,729	9,210	9,047	9,768	-

The data in the table shows that Phoenix always outperforms other tools on all 5 evaluated blockchains. Phoenix covers 9.54%-49.42% more branches because it utilizes the blockchain-specified chaos strategies, which are instrumented in the corresponding context. While other tools only conduct chaos testing at a coarse-grained level without knowing the context of the blockchain system. The results adequately answer **RQ2**.

In addition, we also analyze Phoenix's coverage and identify the top 3 covered components: consensus component, block sync component, and transaction execution component, with 3,462, 1,986, and 1,783 branches on average. While ChaosBlade covers 1,965, 1,529, and 1,623 branches for these components. This demonstrates that the strategies implemented by Phoenix are effective in triggering more execution logic in the corresponding components. However, Phoenix only covered 515 branches in the encryption component on average. This is because a blockchain system may only use a few of the designed encryption methods in a specific configuration. Encryption components are important, but Phoenix currently cannot test more logic in them. We will try to enhance Phoenix with more strategies, such as restarting a node with a different configuration and cover more branches in such components.

6.3 Resilience Issues Reproduction

To evaluate the ability of Phoenix's reproduction, we re-execute the chaos strategy sequences according to the reproduce coordinator's guidance and record the sequence length and reproduction time. We repeated the reproduction experiment multiple times and used average values to eliminate experimental errors. The results are shown in Table 4.

Table 4: The average sequence length and reproduction time required for the Coordinator to reproduce each resilience bug detected by Phoenix.

Bug Number	Sequence Length	Reproduce time (s)
Bug#1	3	157
Bug#2	2	133
Bug#3	10	1413
Bug#4	16	5295
Bug#5	3	103
Bug#6	3	89
Bug#7	5	861
Bug#8	2	537
Bug#9	3	72
Bug#10	2	613
Bug#11	3	381
Bug#12	10	2419
Bug#13	5	1084

From Table 4, we found that Phoenix could successfully reproduce all the 13 detected resilience bugs. The average sequence length of these bugs is 5.15, and the average reproduction time is 1,872.85 seconds. These statistics demonstrate that Phoenix can reproduce these resilience issues accurately and quickly, which adequately answers **RQ3**.

We also found that most resilience issues can be reproduced within 3 sequence steps (Bug #1, Bug #2, Bug #5, Bug #6, Bug #8, Bug #9, Bug #10, and Bug #11). And their reproduction time is relatively short, 260.6 seconds on average. This reveals that most bugs occur at relatively shallow levels, and Phoenix can quickly detect and reproduce them with our context-sensitive chaos strategies. However, some bugs are hidden in deeper logic, such as Bug #3, Bug #4, and Bug #12. They need 10, 16, and 10 sequence steps. Phoenix can also detect and reproduce them, demonstrating that Phoenix can efficiently detect resilience issues in deep paths.

6.3.1 Reproduction case. To help better understand how Phoenix reproduces these resilience bugs, we use one case to illustrate the detailed process. **The case is the bug #4 listed in Table 1**. This bug is a Data Unrecoverable bug that causes some nodes to stop syncing their block data and can not be automatically recovered in the blockchain network. To reproduce this bug, 16 sequence steps

are required for Phoenix. Figure 9 shows the detailed reproduction process. Node 7, node 8, and node 9 are the chaos nodes, and the rest of the nodes are normal nodes.

1st round sequence
$<T_{15}, N_9, H_{51}, S_{9S}>$
$<T_{16}, N_{10}, H_{51}, S_{9S}>$

2nd round sequence
$<T_{00}, N_{10}, H_{12}, S_{1S}>$
$<T_{00}, N_8, H_{51}, S_{9S}>$
$<T_{13}, N_{10}, H_{12}, S_{1E}>$
$<T_{14}, N_8, H_{51}, S_{9E}>$
$<T_{15}, N_9, H_{51}, S_{9S}>$
$<T_{16}, N_{10}, H_{51}, S_{9S}>$

3rd round sequence
$<T_{00}, N_8, H_{12}, S_{1S}>$
$<T_{00}, N_9, H_{12}, S_{1S}>$
$<T_9, N_{10}, H_{12}, S_{1S}>$
$<T_{10}, N_9, H_{12}, S_{1E}>$
$<T_{11}, N_8, H_{12}, S_{1E}>$
$<T_{12}, N_8, H_{51}, S_{9S}>$
$<T_{13}, N_{10}, H_{12}, S_{1E}>$
$<T_{14}, N_8, H_{51}, S_{9E}>$
$<T_{15}, N_9, H_{51}, S_{9S}>$
$<T_{16}, N_{10}, H_{51}, S_{9S}>$

4th round sequence
$\boldsymbol{<T_1, N_8, H_{19}, S_{5S}>}$
$\boldsymbol{<T_2, N_{10}, H_{26}, S_{6S}>}$
$<T_3, N_8, H_{19}, S_{5E}>$
$\boldsymbol{<T_4, N_9, H_{26}, S_{6S}>}$
$\boldsymbol{<T_5, N_8, H_{12}, S_{1S}>}$
$<T_6, N_{10}, H_{26}, S_{6E}>$
$\boldsymbol{<T_7, N_9, H_{12}, S_{1S}>}$
$<T_8, N_9, H_{26}, S_{6E}>$
$\boldsymbol{<T_9, N_{10}, H_{12}, S_{1S}>}$
$<T_{10}, N_9, H_{12}, S_{1E}>$
$<T_{11}, N_8, H_{12}, S_{1E}>$
$\boldsymbol{<T_{12}, N_8, H_{51}, S_{9S}>}$
$<T_{13}, N_{10}, H_{12}, S_{1E}>$
$<T_{14}, N_8, H_{51}, S_{9E}>$
$\boldsymbol{<T_{15}, N_9, H_{51}, S_{9S}>}$
$\boldsymbol{<T_{16}, N_{10}, H_{51}, S_{9S}>}$

S1: Block polluted
S2: Block drop
S3: Transaction Polluted
S4: Transaction drop
S5: Malicious message
S6: Duplicate message
S7: Message drop
S8: Message Delay
S9: Connection break

Figure 9: Reproduction steps of Bug #4. The coordinator selects 2, 6, 10, and 16 steps for each round to reproduce it.

To reproduce this bug, the reproduce coordinator first selects the 2 latest strategies from the strategy pool in the first reproduction round, as shown in Figure 9. Then it checks whether there is an ending strategy without a corresponding starting strategy. Since both of them are starting strategies, no initial strategy needs to be appended. Because $T_{15} < T_{16}$, the coordinator sends a signal *SIGREST* to all other nodes and sends a signal *SIG-H51S9* to node 9. After the connection break strategy is finished, the coordinator will receive a signal *SIGDONE* from node 9. Then the coordinator sends a signal *SIG-H51S9* to node 10 and asks other nodes to stop their chaos strategies. In the meantime, the result checker monitors whether the nodes trigger the bug #4 again.

After the result checker finds that bug #4 is not triggered successfully, the Phoenix coordinator starts the second reproduction round and selects the 4 latest strategies from the strategy pool. Since the first two strategies $< T_{13}, N_{10}, H_{12}, S_{1E} >$ and $< T_{14}, N_8, H_{51}, S_{9E} >$ are the ending strategies, the coordinator appends two starting strategies at the beginning of the sequences. Then it begins the strategies re-execution. Similarly, this bug is not reproduced until the fourth round. In the fourth reproduction round, the Phoenix coordinator schedules the nodes to re-execute the 16 latest strategies based on their timestamps. Finally, this bug has been successfully reproduced by Phoenix. In fact, after our manual analysis, nine of the most important steps are needed to reproduce the bug. Step 1, node 1 first broadcasts a malicious block syncing message to the normal nodes. Steps 2-3, then node 9 and node 10 repeatedly send this malicious message. Steps 4-6, node 8, node 9, and node 10 pollute their block data after other normal nodes ask for syncing their nodes. Steps 6-9, node 8, node 9, and node 10 break their connection during the syncing process. After these steps, normal nodes mistakenly handle such chaotic situations and stop syncing block data from other nodes, which cannot be recovered automatically.

6.4 Resilience Improvement

The resilience of the blockchain systems can be measured by chaos resist time T_1 and chaos recovery time T_2. To evaluate whether the resilience of blockchain systems under test improves or not after fixing the resilience issues detected by Phoenix, we ran Phoenix on the 5 target blockchain systems before and after their fixes. We recorded the number of detected bugs and the recovery time T_2 under different chaos testing duration T_1. We repeated the experiments multiple times and used average values to eliminate experimental errors. The results are shown in Table 5.

Table 5: The average number of detected bugs and the recovery time T_2 under different chaos testing duration T_1 before the bug fixes and after the bug fixes.

Before The Bug Fixes						
T1 (min)	10	20	40	60	80	100
Number of Bugs	6	9	11	13	13	13
T2 (min)	≥10	≥10	≥10	≥10	≥10	≥10
After The Bug Fixes						
T1 (min)	10	20	40	60	80	100
Number of Bugs	0	0	0	0	0	0
T2 (min)	1.71	2.23	2.93	3.97	4.58	5.79

From the first fourth rows of Table 5, we can find that before these resilience bugs are fixed, the number of detected bugs increases as the chaos testing time T_1 grows. Most bugs are triggered within 20 min chaos testing (Bug #1, Bug #2, Bug #5, Bug #8, Bug #9, and Bug #11). This is because they are at the code's shallow levels, which Phoenix can detect quickly. However, the bugs hidden in deeper paths, such as #Bug 3, #Bug 4, and #Bug 12, can also be detected when the chaos testing time T_1 increases. The recovery time T_2 is always larger than 2 hours because all these resilience bugs can not be recovered automatically.

After all these resilience bugs were fixed, no new bugs were found in 60 minutes of chaos. Blockchains under test can always recover automatically. The average recovery time increases slightly when the chaos testing time T_1 grows. They are 1.71, 2.23, 2.93, 3.97, 4.58 and 5.79 minutes, respectively. After fixing the resilience bugs reported by Phoenix, we found that the number of bugs decreased rapidly, from 13 to 0. The average resisting time increases 143.9%, from 24.6 minutes to $\geq$ 60 minutes. Since the blockchain cannot recover before the bugs are fixed, the recovery time is declined from infinite to 2.71 minutes. The declined ratio is unbounded. These statistics demonstrate that Phoenix can effectively improve the resilience of blockchains, which adequately answers **RQ4**.

In practice, it is recommended to set T_{chaos} to 60 minutes and $T_{recover}$ to 10 minutes for the 5 evaluated blockchains (HyperLedger Fabric, Quorum, FISCO-BCOS, Go-Ethereum and BSC). This is because a lower T_{chaos} may miss some bugs. And setting T_{chaos} to 80 minutes and 100 minutes yields the same 13 bugs as the 60-minute setting. Regarding $T_{recover}$, the recovery process typically takes between 1.71 to 5.79 minutes. To minimize false positives, a $T_{recover}$ of 10 minutes is preferred. For another blockchain, it is suggested to set T_{chaos} starting from 10 minutes and increase it until no new bugs are found. The recommended $T_{recover}$ is also 10 minutes since the $T_{recover}$ is determined by recovery mechanisms, which are generally similar in various blockchains.

7 DISCUSSION

7.1 Generality and Flexibility of Phoenix

Phoenix is general and flexible among other systems. Phoenix works with the following steps: 1) A developer injects the context-sensitive strategies into the proper positions in the source code of the blockchain systems. 2) Phoenix uses the test coordinator to control the testing time and checking time for constant chaos testing. 3) The resilience issue checker checks whether the system has bugs. 4) If a bug occurs, Phoenix collects executed strategies into a pool. 5) Phoenix selects the latest 2^n strategies to reproduce the bug. 6) Phoenix checks whether the bug is reproduced successfully by comparing the state and the log.

Only Step 1 is hand-crafted. Developers need to use their system-specific knowledge to locate where to inject Phoenix's strategies. For example, the developers need to find out where the system will send a consensus packet. Based on these positions, developers should inject context-sensitive strategies into Phoenix's utility library. While Steps 2-6 are all automated. With the well-instrumented code, Phoenix automatically performs chaos testing and issue checking. If a bug occurs, Phoenix automatically generates the reproduce sequence and replays the bug.

Designing a new strategy or sampling distribution is easy.

(1) To design a new type of strategy, developers can summarize the lessons learned from historical error situations. Specifically, they can track the history issues and understand where the systems are prone to failure and how they were triggered. Based on these experiences, developers can design a new strategy by implementing a new function that imitates such situations. Based on how they are triggered, developers may know where to inject these functions. These functions can be added to the Phoenix utility library for scheduling by the coordinator.
(2) To add a new sampling distribution, a developer can change the chaos triggering probability according to the particular demands(e.g., component criticality) in the practical usage. Currently, Phoenix's distribution of strategies in a sequence aligns with the actual behaviors of blockchains. If consensus behaviors are more frequent, then Byzantine strategies may have a broader distributions. If developers would like to change the distribution, for example, to reduce the distribution of byzantine strategies. He can add a probability controller in related positions, such as using a random number to control the byzantine strategies should be executed in only 5% situations. In that way, developers can easily change the sampling distribution.

7.2 Unsupported Bug Types

Phoenix does not support bugs caused by non-deterministic inputs. Such inputs like heat fluctuation may impact the hardware environment and trigger a bug. For example, in July 2022, Texas' continuous heat wave impacted the Bitcoin miners seriously [70]. As a result, the Bitcoin system encountered the biggest computing power drop in 2022. Phoenix cannot detect such bugs because it lacks strategies to imitate such non-deterministic inputs. In addition, some logic bugs like consensus unfair [14] are not supported by Phoenix too. Consensus fairness represents that each node's chance of being chosen as a leader or miner should be fair. A bug that may violate this feature is called a consensus unfair bug [9]. However, Phoenix does not support the detection of such bugs due to the lack of corresponding oracles. If enhanced with such oracles, Phoenix can detect them too.

The reproduction process in Phoenix is based on an insight that the resilience issues detected by Phoenix are caused by its chaos strategies (data pollution and deletion, byzantine attacks, and node partition) which are irrelevant to the non-deterministic input sources such as timing and IO interrupts. Thus, Phoenix successfully reproduced all the bugs in our evaluation. However, Phoenix may not support reproducing failures caused by non-deterministic inputs, as described above. We will explore the reproduction method for these failures in the future.

7.3 Strategy Limitations

In our current implementation, Phoenix contains three blockchain-specific strategies. According to our evaluation, Phoenix outputs all issues that other tools detect. This somehow demonstrates that Phoenix's strategies are effective for resilience issues detection. However, there may also be more effective strategies for testing. For example, Phoenix only removes the data files or appends dirty data to them for implementing block/transaction data corruption. By combining these two strategies, Phoenix can also achieve modifying data. However, such modifications are generally more onerous. Minor modifications to data via bit flipping can also be useful for data pollution. In addition, byzantine strategies in Phoenix may modify the timestamp fields of consensus packets to cover part of the time/timestamp pollution situations. While time-servers' failure may also be an effective strategy. Furthermore, hardware-related strategies should be effective in detecting bugs triggered by physical environment fluctuation such as heat. We will attempt to add these strategies to enhance Phoenix.

8 RELATED WORK

Chaos Testing. Chaos testing is considered a useful technique for improving the resilience of distributed systems. Developers in Netflix proposed this idea and created Chaos Monkey [52], which evolved into Simian Army [53] later. Afterward, Alibaba open-sourced its chaos testing tool ChaosBlade [7], which supports multiple chaos experiment types, including CPU controlling, disk occupation, etc. Jepsen [42] is a chaos engineering tool for distributed databases, consensus systems, and queues. Vendors can make accurate claims through a domain-specific language and test their software rigorously. While Hermes [49] injects both context-free and context-dependent faults, such as CPU loading and packet header corruption, into BFT systems. All these tools provide a powerful framework to conduct chaos experiments on distributed systems.

As for blockchain systems, many researchers have used chaos engineering to assess the performance of blockchains. Zhang, Long, et al. propose CHAOSETH [73] to observe application-level metrics such as memory used and disk read of two Ethereum clients under non-blockchain-specific errors like memory filling. While Sondhi, Shiv, et al. [63] apply chaos engineering to evaluate the throughput and latency of various blockchain consensus algorithms. Similarly, BlockBench [15] evaluates the latency and scalability of blockchains by performing network delays, etc.

Blockchain Vulnerability Detection. Plenty of work focuses on detecting the vulnerabilities in blockchain systems too. Generally, they can be divided into two types. The first type is the vulnerability detection of smart contracts. Many tools have been developed to find bugs in solidity contracts, such as reentrancy, overflow, transaction order dependency, etc. For example, Oyente [44] and Pluto [48] use symbolic execution to construct the control flow graph of a contract and analyze it. While tools such as sFuzz [54], ILF [34], and V-Gas [47] use fuzzing techniques to generate multiple inputs to the contract functions and trigger the hidden bugs. Researchers also develop tools based on datalog analysis for smart contact vulnerability detection. For example, Securify [66] establishes datalog rules based on contract IR. While Pied-Piper [46] detects backdoors in smart contracts by both datalog analysis and fuzzing. Ren, M, et al. [60] made an empirical study on these tools and proposed SCStudio [59] which integrates them to increase the detection accuracy.

The other type of blockchain vulnerability is the one in the underlying platform. Many researchers focus on the security of the key components of blockchains. For example, Fluffy [71] is a tool that generates multiple transactions to test the consensus process of Ethereum. While EVMLab [20] and EVMFuzzer [31] focus on the flaws of the Ethereum virtual machine. In addition, Tyr [9] and LOKI [45] use fuzzing techniques to detect the implementation bugs in consensus protocols of blockchain systems.

Similarly, Twins [1] is a unit-test generator to test BFT implementations for Diem blockchain, which also performs byzantine attacks on blockchains. However, according to Twins's source code and Winter, Levin N., et al. [69], Twins's implementation is tightly bound to the DiemBFT, and porting Twins would likely require a full reimplementation. Thus, we only compare it with Phoenix on Diem. Specifically, we implement Phoenix on Diem by the following steps: 1) We identify the positions in the Diem's source code where the consensus packets are sent, and the block data are synchronized. 2) We inject the strategies defined in the Phoenix utility library in these positions. 3) We add the issue checkers by analyzing the log of Diem nodes. As a result, Phoenix successfully detected a node unrecoverable issue [23] , while Twins could not find it. Phoenix triggers this bug by conducting a byzantine strategy before sending consensus packets and then inject a node partition of the non-byzantine nodes. Twins cannot find this bug because it lacks consideration of context information, while Phoenix supports more strategies and instruments them based on the chain context.

Main Difference. In this paper, Phoenix mainly focuses on the resilience issue detection of blockchain systems. Unlike traditional chaos testing tools, Phoenix develops context-sensitive blockchain-specific chaos strategies. As for other chaos tools targeted at blockchain, they lack resilience issue checkers and context-sensitive strategies. Thus, they cannot perform chaos testing effectively. In addition, without coordinating their strategies, they cannot reproduce the chaos experiment steadily like Phoenix. As for other blockchain vulnerability detection work, they have different goals from Phoenix. Smart contract vulnerability detection tools target the application level, while Phoenix focuses on the underlying level. Tools like Fluffy, EVMLab, and EVMFuzzer try to detect bugs in the contract execution process. LOKI, Tyr, and Twins target the implementation of consensus protocols. While Phoenix mainly focuses on resilience vulnerabilities that lead to node or data unrecoverable.

9 CONCLUSION

In this paper, we propose Phoenix, a system to detect and locate resilience issues in blockchains via context-sensitive chaos. First, we design three context-sensitive chaos strategies. Then we use coordinators and issue checkers to perform the chaos testing and recovery phases. For the detected issues, Phoenix generates reproduction sequences to locate them. We implement and evaluate Phoenix on 5 commercial blockchain systems. Phoenix successfully detected 13 previously unknown resilience bugs. Besides, Phoenix successfully reproduces all the detected bugs with 5.15 sequence steps on average. After fixing the resilience bugs, the average chaos resists time T_1 is increased by 143.9%. Our future work will focus on applying Phoenix on other systems with more chaos strategies.

10 ACKNOWLEDGEMENT

This research is sponsored in part by the National Key Research and Development Project (No.2022YFB3104000), NSFC Program (No. 62022046, 92167101, U1911401, 62021002, U20A6003), and Webank Scholar Project (20212001829)

REFERENCES

[1] Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, and Dahlia Malkhi. Twins: White-glove approach for bft testing. *arXiv preprint arXiv:2004.10617*, 2020.
[2] bnb chain. Bnb smart chain. https://github.com/bnb-chain/bsc/releases/tag/v1.1.17, 2023. Accessed at April 23, 2023.
[3] Tammy Butow. A brief history of chaos engineering. https://www.gremlin.com/community/tutorials/chaos-engineering-the-history-principles-\and-practice/#a-brief-history-of-chaos-\engineering, 2022. Accessed at December 6, 2022.
[4] Gabriel R Carrara, Leonardo M Burle, Dianne SV Medeiros, Célio Vinicius N de Albuquerque, and Diogo MF Mattos. Consistency, availability, and partition tolerance in blockchain: a survey on the consensus mechanism over peer-to-peer networking. *Annals of Telecommunications*, 75(3):163–174, 2020.
[5] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.
[6] BNB Chain. Bnb smart chain. https://www.bnbchain.org/en/smartChain, 2022. Accessed at December 23, 2022.
[7] chaosblade io. Chaos blade. https://netflix.github.io/chaosmonkey/, 2022. Accessed at December 29, 2022.
[8] JP Morgan Chase. Quorum white paper. *Accessed: Jan*, 17:2019, 2016.
[9] Yuanliang Chen, Fuchen Ma, Yuanhang Zhou, Yu Jiang, Ting Chen, and Jiaguang Sun. Tyr: Finding consensus failure bugs in blockchain system with behaviour divergent model. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1186–1201. IEEE Computer Society, 2022.
[10] ConsenSys. Goquorum. https://github.com/ConsenSys/quorum, 2022. Accessed at December 6, 2022.
[11] Curve. Curve swap. https://curve.fi/#/ethereum/swap, 2022. Accessed at December 6, 2022.
[12] NATIONAL VULNERABILITY DATABASE. Cve-2021-35041 detail. https://nvd.nist.gov/vuln/detail/CVE-2021-35041, 2022. Accessed at December 23, 2022.
[13] NATIONAL VULNERABILITY DATABASE. Cve-2021-43669 detail. https://nvd.nist.gov/vuln/detail/CVE-2021-43669, 2022. Accessed at December 23, 2022.
[14] NATIONAL VULNERABILITY DATABASE. Cve-2022-28937 detail. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-28937, 2022. Accessed at July 18, 2023.
[15] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. Blockbench: A framework for analyzing private blockchains. In *Proceedings of the 2017 ACM international conference on management of data*, pages 1085–1100, 2017.
[16] Clang 13 documentation. Address sanitizer. https://clang.llvm.org/docs/AddressSanitizer.html, 2022. Accessed at December 6, 2022.
[17] Gcov documentation. A test coverage program. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, 2022.
[18] GoogleTest documentation. Googletest coverage. https://github.com/google/googletest, 2022.
[19] drandreaskrueger. Chainhammer ethereum benchmarking. https://github.com/drandreaskrueger/chainhammer, 2022. Accessed at December 28, 2022.
[20] Ethereum. Evmlab. https://github.com/ethereum/evmlab, 2020.

[21] Ethereum. Welcome to ethereum. https://ethereum.org/en/, 2022. Accessed at December 23, 2022.

[22] Fabric. https://github.com/SmartBFT-Go/fabric, 2022. Accessed at December 6, 2022.

[23] fcorleone. The blockchain stucks when there are some malicious nodes. https://github.com/diem/diem/issues/10228, 2022. Accessed at July 18, 2023.

[24] fcorleone. A malicious node may fake a proposal's header. https://github.com/FISCO-BCOS/FISCO-BCOS/issues/2307, 2022. Accessed at July 18, 2023.

[25] FISCO. Fisco bcos. https://github.com/FISCO-BCOS/FISCO-BCOS, 2022. Accessed at December 6, 2022.

[26] FISCO. Stress testing guidelines. https://fisco-bcos-doc.readthedocs.io/zh_CN/latest/docs/develop/stress_testing.html, 2022. Accessed at December 28, 2022.

[27] FISCO-BCOS. Fisco bcos. https://github.com/FISCO-BCOS/FISCO-BCOS, 2022. Accessed at December 6, 2022.

[28] JAKE FRANKENFIELD. Proof of stake. https://www.investopedia.com/terms/p/proof-stake-pos.asp, 2021. Accessed at April 23, 2023.

[29] JAKE FRANKENFIELD. Proof of work. https://www.investopedia.com/terms/p/proof-work.asp, 2021. Accessed at April 23, 2023.

[30] JAKE FRANKENFIELD. Hard fork: What it is in blockchain, how it works, why it happens. https://www.investopedia.com/terms/h/hard-fork.asp, 2022. Accessed at December 6, 2022.

[31] Ying Fu, Meng Ren, Fuchen Ma, Heyuan Shi, Xin Yang, Yu Jiang, Huizhong Li, and Xiang Shi. Evmfuzzer: detect evm vulnerabilities via fuzz testing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1110–1114, 2019.

[32] go ethereum. go-ethereum. official go implementation of the ethereum protocol. https://geth.ethereum.org/, 2022. Accessed at December 6, 2022.

[33] HamidullahMuslih. Geth restarting : panic: runtime error: invalid memory address or nil pointer dereference. https://github.com/ethereum/go-ethereum/issues/26338, 2022. Accessed at April 20, 2023.

[34] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 531–548, 2019.

[35] Frank Hofmann, Simone Wurster, Eyal Ron, and Moritz Böhmecke-Schwafert. The immutability concept of blockchains and benefits of early standardization. In *2017 ITU Kaleidoscope: Challenges for a Data-Driven Society (ITU K)*, pages 1–8. IEEE, 2017.

[36] holiman. panic in findbeaconancestor. https://github.com/ethereum/go-ethereum/issues/25787, 2022. Accessed at April 20, 2023.

[37] holiman. panic on geth/downloader/beaconsync.go:220. https://github.com/ethereum/go-ethereum/issues/26300, 2022. Accessed at April 20, 2023.

[38] Hyperledger. Hyperledger fabric. https://www.hyperledger.org/use/fabric, 2022. Accessed at December 6, 2022.

[39] Hyperledger. Hyperledger fabric. https://github.com/hyperledger/fabric/tree/release-2.3, 2022. Accessed at December 6, 2022.

[40] hyperleger. Hyperledger fabric samples. https://github.com/hyperledger/fabric-samples, 2022. Accessed at December 28, 2022.

[41] Mubashar Iqbal and Raimundas Matulevičius. Exploring sybil and double-spending risks in blockchain systems. *IEEE Access*, 9:76153–76177, 2021.

[42] Jepsen. Distributed systems safety research. https://jepsen.io/, 2023. Accessed at April 23, 2023.

[43] logo. Configure qbft consensus. https://consensys.net/docs/goquorum/en/latest/configure-and-manage/configure/consensus-protocols/qbft/, 2022. Accessed at December 6, 2022.

[44] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. *IACR Cryptology ePrint Archive*, page 633, 2016.

[45] Fuchen Ma, Yuanliang Chen, Meng Ren, Yuanhang Zhou, Yu Jiang, Ting Chen, Huizhong Li, and Jiaguang Sun. Loki: State-aware fuzzing framework for the implementation of blockchain consensus protocols. In *Proceedings 2023 Network and Distributed System Security Symposium*, 2023.

[46] Fuchen Ma, Meng Ren, Lerong Ouyang, Yuanliang Chen, Juan Zhu, Ting Chen, Yingli Zheng, Xiao Dai, Yu Jiang, and Jiaguang Sun. Pied-piper: Revealing the backdoor threats in ethereum erc token contracts. *ACM Transactions on Software Engineering and Methodology*, 2022.

[47] Fuchen Ma, Meng Ren, Fu Ying, Wanting Sun, Houbing Song, Heyuan Shi, Yu Jiang, and Huizhong Li. V-gas: Generating high gas consumption inputs to avoid out-of-gas vulnerability. *ACM Transactions on Internet Technology (TOIT)*, 2018.

[48] Fuchen Ma, Zhenyang Xu, Meng Ren, Zijing Yin, Yuanliang Chen, Lei Qiao, Bin Gu, Huizhong Li, Yu Jiang, and Jiaguang Sun. Pluto: Exposing vulnerabilities in inter-contract scenarios. *IEEE Transactions on Software Engineering*, 48(11):4380–4396, 2021.

[49] Rolando Martins, Rajeev Gandhi, Priya Narasimhan, Soila Pertet, António Casimiro, Diego Kreutz, and Paulo Veríssimo. Experiences with fault-injection in a byzantine fault-tolerant protocol. In *Middleware 2013: ACM/IFIP/USENIX 14th International Middleware Conference, Beijing, China, December 9-13, 2013, Proceedings 14*, pages 41–61. Springer, 2013.

[50] mikoim. findbeaconancestor: panic: runtime error: invalid memory address or nil pointer dereference. https://github.com/ethereum/go-ethereum/issues/26020, 2022. Accessed at April 20, 2023.

[51] CVE Mitre. Cve-2021-43668. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-43668, 2022. Accessed at July 18, 2023.

[52] Netflix. Chaos monkey. https://netflix.github.io/chaosmonkey/, 2022. Accessed at December 29, 2022.

[53] Netflix. Simian army. https://github.com/Netflix/SimianArmy, 2022. Accessed at December 29, 2022.

[54] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. *arXiv preprint arXiv:2004.08563*, 2020.

[55] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.

[56] Ethereum Org. The beacon chain. https://ethereum.org/en/roadmap/beacon-chain/, 2023. Accessed at April 20, 2023.

[57] Ethereum Org. Nemata (v1.10.25). https://github.com/ethereum/go-ethereum/releases/tag/v1.10.25, 2023. Accessed at April 23, 2023.

[58] Ethereum Org. Skeleton. https://github.com/ethereum/go-ethereum/blob/722bb210bfe86984b39c80dcab79405157338f25/eth/downloader/skeleton.go, 2023. Accessed at April 20, 2023.

[59] Meng Ren, Fuchen Ma, Zijing Yin, Ying Fu, Huizhong Li, Wanli Chang, and Yu Jiang. Making smart contract development more secure and easier. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1360–1370, 2021.

[60] Meng Ren, Zijing Yin, Fuchen Ma, Zhenyang Xu, Yu Jiang, Chengnian Sun, Huizhong Li, and Yan Cai. Empirical evaluation of smart contract testing: What is the best choice? In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 566–579, 2021.

[61] ricardolyn. downloader: segmentation fault on restart when downloading beacon headers. https://github.com/ethereum/go-ethereum/issues/26764, 2022. Accessed at April 20, 2023.

[62] rjl493456442. eth/downloader: fix unexpected skeleton header deletion. https://github.com/ethereum/go-ethereum/pull/26451, 2022. Accessed at April 20, 2023.

[63] Shiv Sondhi, Sherif Saad, Kevin Shi, Mohammad Mamun, and Issa Traore. Chaos engineering for understanding consensus algorithms performance in permissioned blockchains. In *2021 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*, pages 51–59. IEEE, 2021.

[64] Dean Steinbeck. Crypto snippets: The stellar blockchain just crashed — this is why nodes matter. https://cryptolawinsider.com/stellar-crash/, 2023. Accessed at January 9, 2023.

[65] Tether. Tether token. https://tether.to/en/, 2022. Accessed at December 6, 2022.

[66] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin T. Vechev. Securify: Practical security analysis of smart contracts. In *ACM Conference on Computer and Communications Security*, 2018.

[67] Ingo Weber, Vincent Gramoli, Alex Ponomarev, Mark Staples, Ralph Holz, An Binh Tran, and Paul Rimba. On availability for blockchain-based systems. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 64–73. IEEE, 2017.

[68] WikiPedia. Linear temporal logic. https://en.wikipedia.org/wiki/Linear_temporal_logic, 2022. Accessed at December 6, 2022.

[69] Levin N Winter, Florena Buse, Daan De Graaf, Klaus Von Gleissenthall, and Burcu Kulahcioglu Ozkan. Randomized testing of byzantine fault tolerant algorithms. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):757–788, 2023.

[70] RACHEL WOLFSON. Texas a bitcoin 'hot spot' even as heat waves affect crypto miners. https://cointelegraph.com/news/texas-a-bitcoin-hot-spot-even-as-heat-waves-affect-crypto-miners, 2023. Accessed at July 18, 2023.

[71] Youngseok Yang, Taesoo Kim, and Byung-Gon Chun. Finding consensus bugs in ethereum via multi-transaction differential fuzzing. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 349–365, 2021.

[72] Guorui Yu, Shibin Zhao, Chao Zhang, Zhiniang Peng, Yuandong Ni, and Xinhui Han. Code is the (f) law: Demystifying and mitigating blockchain inconsistency attacks caused by software bugs. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*, pages 1–10. IEEE, 2021.

[73] Long Zhang, Javier Ron, Benoit Baudry, and Martin Monperrus. Chaos engineering of ethereum blockchain clients. *arXiv preprint arXiv:2111.00221*, 2021.

[74] Xu Zhao, Zhiwei Lei, Guigang Zhang, Yong Zhang, and Chunxiao Xing. Blockchain and distributed system. In *International Conference on Web Information Systems and Applications*, pages 629–641. Springer, 2020.