Chord: Towards a Unified Detection of Blockchain Transaction Parallelism Bugs

Yuanhang Zhou*†, Zhen Yan*†, Yuanliang Chen†, Fuchen Ma†✉, Ting Chen‡, and Yu Jiang†✉
†BNRist, Tsinghua University, Beijing, China
‡University of Electronic Science and Technology of China, Chengdu, China

***Abstract*—Blockchain systems have implemented various transaction parallelism mechanisms to improve the system throughput and reduce the latency. However, they inevitably introduce bugs. Such bugs can result in severe consequences such as asset loss, double spending, consensus failure, and DDoS. Unfortunately, they have been little analyzed about their symptoms and root causes, leading to a lack of effective detection methods.**

In this work, we conduct a thorough analysis of historical transaction parallelism bugs in four commercial blockchains. Results show that most of them arise from mishandling conflicting transactions and manifest without obvious phenomena. However, given the heterogeneity of blockchains, it is challenging to trigger conflict handling in a unified way. Effectively identifying these bugs is also hard. Inspired by the findings, we propose *Chord*, aiming at detecting blockchain transaction parallelism bugs. *Chord* proposes a unified conflict transaction model to generate various conflict transactions. *Chord* also dynamically adjust the transaction submission and inserts proactive reverts during transaction execution to conduct thorough testing. Besides, *Chord* incorporates a local-remote differential oracle and a TPS oracle to capture the bugs. Our evaluation shows that *Chord* successfully detects 54 transaction parallelism bugs. Besides, *Chord* outperforms the existing methods by decreasing the TPS by 49.7% and increasing the latency by 388.0%, showing its effectiveness in triggering various conflict scenarios and exposing the bugs.

I. Introduction

Today, the performance of traditional blockchain systems struggles to keep up with the growing throughput demands. Specifically, for time-sensitive applications like stock exchange [3], [4], [1], the throughput and latency of blockchain systems have consistently presented a primary bottleneck. To address this problem, early in 2020, Ethereum [11] has introduced danksharding in its roadmap to enhance the TPS. However, the full implementation is still years away [9]. Hyperledger Fabric [22] parallelizes all transactions and discards conflicting ones after validation, but still exhibits low performance due to many invalid transactions [63], [64]. Therefore, recently emerged commercial blockchains have established various transaction parallelism mechanisms [43], [12], [65], [54], [18]. They schedule the parallel transactions based on their interdependencies to ensure safe and effective parallel execution. For example, FISCO BCOS conducts DAG analysis [12] to identify the conflict transactions, and proposes a DMC Scheduler [13] to parallelize non-conflict ones.

However, these transaction parallelism mechanisms inevitably introduce bugs. We name the bug as TPB (short for **T**ransaction **P**arallelism **B**ug). TPBs compromise the performance, deviating from their design intentions. Moreover, they lead to severe consequences such as asset losses, DDoS, etc. For example, Solana [57] schedules the rent collection transactions by dividing the requested ranges into partitions and executing them in parallel. However, it did not check the conflicts between the ranges. On Jun 16, 2022, when transactions accessed overlapping ranges, the unsafe parallelism resulted in collecting rent from the same account twice [25], leading to asset losses. Unfortunately, TPBs have not been sufficiently analyzed and detected.

To better understand TPB, we conducted a thorough study on 50 historical TPBs in four widely-used commercial blockchains with transaction parallelism mechanisms, including FISCO BCOS [14], Sei [53], Solana [57], and Aptos [38]. We found that: ① 82.0% of the TPBs arise in the conflict transaction handling process. Blockchains are error-prone when analyzing and scheduling trasnactions accessing composite data types. Besides, unexpected transaction reverts and exceptions usually result in TPBs. ② The TPBs bring catastrophic but inconspicuous consequences. 66% of them result in incorrect execution outcomes, which can lead to asset losses. The others result in an unexpected decrease in throughput, triggering DDoS attacks or even system crashes.

Existing works cannot detect TPBs effectively. Traditional concurrency bug detectors [41], [44], [68], [56], [40], [66] cannot detect TPBs because they focus on scheduling threads to trigger data races. But TPBs arise from bugs in transaction parallelism mechanisms, and require parallel conflict transactions to trigger them (finding ①). For transaction generation tools, Fluffy [67] sends sequential transactions and checks the consistency among nodes. EVMFuzzer [17] and EVMLab [10] mutate the smart contracts and generate a single transaction. Tools for front-running vulnerability [71], [59], [70] generate transaction sequences accessing profitable variables. However, sequential transactions or a single transaction cannot trigger conflict handling process (finding ①). Besides, some TPBs hold unique triggering conditions on the transaction submission and execution stage (finding ①), which are ignored in existing works. Furthermore, the non-determinism of parallelism introduces unique challenges for capturing TPBs, so there are no well-defined oracles for TPB scenarios (finding ②).

The first challenge is how to trigger the conflict handling effectively for various transaction parallelism mechanisms. According to our finding ①, most of the TPBs arise in conflict

* Yuanhang Zhou and Zhen Yan contributed equally to this work.
✉ Fuchen Ma and Yu Jiang are the corresponding authors.

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00205

handling processes. But the diversity of the blockchain structures and mechanisms make it challenging to cover various conflict scenarios in a general way. Furthermore, transactions may be interrelated in different ways. Some of them access the same shared objects. Others hold indirect dependencies. Exceptions such as out-of-gas and runtime exceptions also lead to new conflict relations. Therefore, it's challenging to generate high-quality conflict transactions that can trigger various error-prone conflict handling scenarios.

The second challenge is how to effectively identify TPBs when they occur, given their inconspicuous nature. According to our finding ②, most TPBs do not manifest obvious symptoms like node crashes or panics. In contrast, they usually lead to some silent consequences such as incorrect execution results. Thus, identifying such scenarios is a challenging task. The transaction parallelism mechanisms introduce non-determinism regarding the execution order. Besides, some transactions may be forcibly reverted or discarded due to conflicts. Therefore, it is hard to determine the expected results solely based on the content of the submitted transactions.

To address the aforementioned challenges, we propose *Chord*. To resolve the first challenge, *Chord* establishes a unified conflict transaction model for triggering various conflict handling scenarios. *Chord* first proposes a template contract consisting of various shared resources and access operations. Based on it, *Chord* applies the conflict constructor to generate transactions with complex conflict relations. *Chord* dynamically adjusts the transaction submission timing to trigger conflicts more frequently. Besides, *Chord* applies the revert injector to trigger more TPB-prone cases during transaction execution. To resolve the second challenge, *Chord* adopts a local-remote differential oracle to reproduce the expected results locally and identify the inconsistency of on-chain results with the expected ones. Furthermore, *Chord* introduces a TPS oracle to identify TPBs that lead to the abnormal throughput decrease. In this way, *Chord* can detect TPB effectively.

We evaluated *Chord* on four widely-used commercial blockchains with different transaction parallelism mechanisms. *Chord* successfully detects 54 TPBs, including 10 previously unknown ones. In comparison, equipped with *Chord*'s oracles, existing tools only detect 7 TPBs. Besides, we compared *Chord* with existing tools on the TPS and latency to evaluate whether *Chord* triggers the conflict handling processes effectively. The decline in TPS and the rise in latency imply an increase in conflicts among transactions, thereby indicating a higher probability of triggering TPBs (according to finding ①). Results show that *Chord* decreases the TPS by 14.80%-93.66%, and increases the latency by 12.44%-1205.59%, showing that *Chord* generates high-quality conflict handling scenarios and effectively triggers TPBs.

This paper makes the following contributions:

- We studied the symptoms and root causes of 50 historical TPBs in four commercial blockchains.
- we proposed *Chord*. *Chord* establishes the unified conflict transaction model and two dedicated TPB oracles.
- We evaluated *Chord* on 4 blockchains. *Chord* detects 54 TPBs including 10 previously unknown ones. We have open-sourced *Chord* for practical usage.

II. Background

Blockchain is a decentralized system consisting of multiple nodes, where users can deploy smart contracts containing functions that can be invoked through transactions. Transactions trigger predefined actions such as transferring funds, storing data, or executing logic within the smart contract. Previously, transactions are executed serially by each node, ensuring security and determinism but with limited performance, which cannot meet the demands of high-throughput applications.

To address this, various transaction parallelism mechanisms have been proposed, allowing transactions to be executed in parallel, thereby improving throughput and reducing latency. Secure parallelism relies on identifying and scheduling conflicting transactions, which occur when two or more transactions attempt to access or modify the same resource. Conflicts arise when one transaction reads and another writes to the same resource, or when multiple transactions try to modify it simultaneously. Several approaches have been proposed to handle transaction conflicts. Some require users to specify all shared variables accessed by a transaction, followed by static analysis to detect conflicts. For example, FISCO BCOS 2.0 [12] requires users to pre-declare accessed variables and uses DAG analysis to identify conflicts, while Solana's Sealevel [65] also asks users to specify read/write relationships before submission. Other mechanisms, like FISCO BCOS 3.0's DMC scheduler [13], perform conflict analysis autonomously. Some systems, such as Sei, employ optimistic parallelization [54], executing all transactions in parallel and re-running conflicting ones sequentially. Aptos uses Block-STM [18] to optimistically execute pre-ordered transactions and resolve conflicts by rectifying memory access errors.

While transaction parallelism mechanisms significantly improve blockchain performance, their implementation can introduce bugs, especially when handling complex conflicting transactions, leading to errors with serious consequences.

III. Motivating Example

TPB has become widespread and hard to detect. But once occurs, it can lead to severe consequences. An example is a TPB in Solana's transaction parallelism mechanism, triggered in a built-in module *bank*. The *bank* tracks client accounts and the progress of on-chain programs.

However, this introduced a TPB. Fig. 1 shows the triggering process of this TPB. When transactions interact with the *bank* in parallel, the accounts they access may have overlaps between 'partitions', which make them conflict. As shown in Fig. 1, tx_1 and tx_2 access the range of [0, 2] and [1, 3] respectively, resulting in an overlap in [1, 2]. Performing read-modify-write operations to the same storage in parallel is unsafe. Unfortunately, Solana overlooks the check for conflicts

Chord at: https://anonymous.4open.science/r/Chord-E070

among these requested 'partitions'. In this case, results val'_1 and val'_2 do not match the expected r and val^e_2. Therefore, this TPB leads to incorrect results, and results in asset losses.

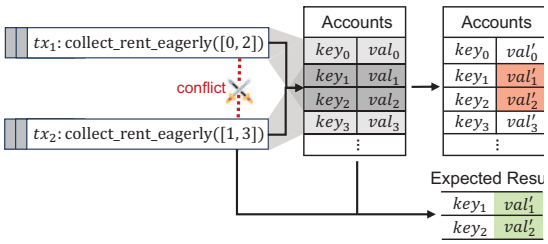


Fig. 1: The triggering process of the TPB in S mishandling of conflict parallel transactions leads calculation results.

To address this TPB, in Solana:pr-25991 [26], fixed the logic of *collect_rent_eagerly*. The code presented in Listing. 1. As shown in line 7-19, wh carries out the rent collection, it first checks all th partitions to determine if there are overlaps betwe an overlap is found, the signal *parallel* will be s and the conflict transactions will be executed seri 24-26. If no overlap is detected, the previous logic of parallel execution is maintained in line 20-22.

From this case, we learn that: 1) TPBs usually lie in conflict handling processes, which require conflict transactions to be triggered. 2) TPBs can result in incorrect execution outcomes, requiring specifically designed oracles. Unfortunately, there is no existing work that can detect such TPBs. Traditional concurrency bug detectors for distributed systems cannot generate transactions and trigger parallel scheduling. Fluffy [67], EVMFuzzer [17] and EVMLab [10] generate single or sequential transactions for testing. Existing works for front-running attacks generate profitable transaction sequences. However, they cannot generate various conflict transactions and frequently trigger transaction parallelism. In this example, they cannot generate conflict transactions that access the buggy function with overlapping ranges. Besides, their oracles cannot deal with the non-determinism of parallel execution, therefore cannot identify this TPB.

To address these challenges, we propose *Chord*. *Chord* establishes a unified conflict transaction model to generate conflict transactions. In this case, *Chord* establishes the template contract that includes the self-defined module *bank*, and provides specialized access approaches *collect_rent_eagerly*. Then *Chord* generates conflict transactions with overlapping ranges for rent collection. *Chord* controls the submission QPS to frequently trigger conflicts. Besides, by reproducing the expected results based on the transaction receipts, *Chord* identifies the incorrect on-chain results and captures the TPB.

IV. Overview of Transaction Parallelism Bug

To understand the features of TPBs and inspire subsequent detection, we conduct an in-depth study on real-world TPBs.

```
1  fn collect_rent_eagerly(&self) {
2    ...
3  - let thread_pool = ...;
4  - thread_pool.install(|| {
5  -     ...
6  - });                          Bug here: directly execute in parallel
7  + if parallel {
8  +     let ranges = partitions.iter()
9  +     ...
10 +     'outer: for i in 0..ranges.len() {
11 +         for j in 0..ranges.len() {
12 +             if i == j { continue; }
13 +             // check the overlap
14 +             if i.contains(j.start()) || i.contains(j.end()) {
15 +                 parallel = false;
16 +                 break 'outer;
17 +             }
18 +         }                      Bug fix: check the overlap first
19 +     }
20 +     if parallel {
21 +         // collect in parallel
22 +     }
23 + }
24 + if !parallel {
25 +     // collect in serial
26 + }}
27   ...
28 }
```

Listing. 1: The code snippet of the TPB Solana:pr-25991 [26]. The TPB was introduced when parallelizing the execution of partitions in Line 3-6. Line 7-26 fixed it by checking the overlaps between partitions.

We now present our study methodology and our findings.

A. Study Methodology

We conduct rigorous empirical analysis following the open-coding method. Our study goes through the following steps:

1. Data Collection: We collect the TPBs from the 4 widely-used blockchains with various transaction parallelism mechanisms, including FISCO BCOS, Sei, Aptos, and Solana. We select TPBs from issue trackers and pull requests of the targeted blockchains. We filter out the valuable items containing keywords 'parallel', 'schedule', 'conflict', the specific name of the parallelism mechanism ('DMC' for FISCO BCOS, 'Sealevel' for Solana, 'Optimistic Parallelization' for Sei, 'Block-STM' for Aptos) in their titles and contents. We refine the selection by retaining items involving assignees, and tags like 'bug', 'security', etc. Finally, we obtain 615 items.

2. Data Familiarization: For each collected item, we extract the existing information provided in its issue or pull request description to further decide if it is a TPB. Totally, we identify 50 TPBs (25 from FISCO BCOS, 15 from Sei, 9 from Solana, 1 from Aptos). 15 TPBs have provided information of attack vectors. 41 TPBs' descriptions reveal their symptoms. This helps us determine the basic nature of each TPB, laying the foundation for further analysis.

3. Initial Coding: In this step, we deeply examine the source code and fix commits to gather more detailed information. For TPBs that have existing descriptions, we study the buggy code and patches to verify the root cause. For TPBs missing information, we directly analyze the source code to investigate their root cause and potential consequences.

4. Categorization: We proceed to systematically classify the TPBs into distinct categories based on their observable symptoms and behaviors. The categorization allows us to identify the patterns and commonalities among various TPBs, facilitating the subsequent analysis on the common triggering conditions. Specifically, we categorize the TPB symptoms into two main types: those that lead to incorrect execution results (33/50) and those that affect system performance (17/50).

5. Hypothesis Formation: Finally, we identify and count the root causes across various TPBs. We focus on the affected variables, the components where TPBs occur, and the execution stages in which they appear. Through statistical analysis, we identify 2 most common triggering conditions to guide subsequent detection. For special cases, we also conduct a detailed analysis to ensure comprehensive study.

B. Symptoms

To study how to capture TPBs, we conduct an in-depth study on their symptoms. Results show that TPBs are often inconspicuous and cannot be easily observed. However, they can lead to severe consequences.

33/50 TPBs result in incorrect execution outcomes. After confirming the transactions, the states reached by the blockchain are not consistent with the expected states. Specifically, they lead to the error calculation of balances (8/33), gas price (2/33), certification verification (2/33), and other global variable status (21/33). For example, Solana:pr-25991 [26] is triggered by two conflicting transactions with overlapping ranges on the rented accounts. Solana's transaction parallelism mechanism fails to identify the conflict conditions, falsely parallel the execution, and finally leads to collecting rent from the same account twice. This TPB causes double spending attacks, which pose threats to the asset safety. Besides, FISCO BCOS:issue-2101 [23] is triggered by massive parallel conflict transactions simultaneously modifying the account balance. Under high workload, the blockchain fails to successfully synchronize the actual execution order and state of the conflict transactions, resulting in discrepancies in user balance values across different node views. These TPBs pose threats to the security of on-chain assets, resulting in asset losses, double spending, etc. However, these symptoms do not lead to evident phenomena such as crashes. Therefore, without a dedicated oracle, capturing these TPBs is challenging.

Besides, 17/50 TPBs affect the system performance, even making the blockchain unable to process transactions. It is usually manifested by an abnormal decline in the blockchain's TPS (transactions per second) or the abnormal increase in the latency. Some TPBs such as Sei:pr-483 [39] and Solana:issue-29895 [19] are caused by mishandling of plenty of parallel conflicting transactions. The parallelism mechanisms falsely schedule the transactions, leading to a large number of transactions being dropped or cannot be processed. They ultimately cause the blockchain system to remain in an incorrect transaction execution state, preventing it from processing any subsequent transactions. Besides, Solana:issue-12441 [36] arises because the blockchain's transaction parallelism mechanism fails to consider the resource consumption when handling parallel transactions. Overloaded transactions ultimately lead to the exhaustion of CPU or memory resources. These TPBs destroy the liveness of blockchain, and may result in DDoS attacks.

C. Root Causes

To trigger TPBs, we conducted a thorough investigation on the root causes and triggering conditions of real-world TPBs. Below, we present our findings.

Trigger 1: *Most TPBs are triggered by parallel conflict transactions accessing composite data types.*

Based on our study, 41/50 TPBs require conflicting transactions to be triggered. The blockchain's transaction parallelism mechanism requires careful design across various execution stages of conflict transactions, including conflict detection, transaction scheduling, execution, and error handling. These complex mechanisms inevitably introduce implementation bugs that lead to TPBs. Specifically, 22 out of 41 TPBs occur when blockchains fail to correctly identify conflicts and schedule conflict transactions. Additionally, 7 out of 41 TPBs are caused by insufficient checks and handling of runtime exceptions during the transaction scheduling process, leading to robustness issues. Furthermore, 12 out of 41 TPBs are triggered by improper handling of reverts during the execution of conflict transactions, resulting in deadlocks or incorrect updates of execution results.

Furthermore, we found that such TPBs are always triggered by conflicting read/write on composite data types, such as pre-defined classes, structures, mappings, or external contracts. This is because they exhibit more intricate read and write operations. For example, FISCO BCOS:issue-3826 [24], FISCO BCOS:pr-3446 [34], FISCO BCOS:pr-2339 [37] are all triggered by conflicting read or write operations on KvTable objects, which is a pre-compiled class that holds specific functions. Solana:pr-25991 [26] also arises in a self-defined class object Bank. Access operations on these composite variables are intricate, involving multi-level function calls and collaborative actions. Consequently, accurately identifying conflicts is more challenging for them. Some other blockchains are implemented in Rust or Move, which have the nature to ensure thread safety. However, they are also vulnerable to complex conflicting scenarios. For example, Sei:pr-637 [8] and Sei:pr-647 [7] are triggered by the mishandling of conflicting transactions, leading to unexpected non-determinism in the execution results. Several TPBs require special triggering conditions. 6 TPBs are triggered by consensus failure, resource misallocation, and incorrect configuration together with the conflict transactions. The remaining 9 are triggered by extensive parallel transactions without the need for conflicts.

Trigger 2: *Most TPBs arise during conflict handling and exception handling in transaction execution process.*

We studied the scenarios where TPBs occur and found that unexpected runtime exceptions and transaction rollbacks during conflict handling are highly prone to triggering TPBs.

Under these conditions, blockchains need to modify the scheduling status of transactions or provide error handling approaches. However, these processes are prone to TPBs. Sei:pr-410 is triggered during rollbacks, when Sei incorrectly reverts only the execution result without reverting the version number of the related shared variable. Consequently, subsequent conflicting transactions are unable to perform further operations on that variable. Similarly, in FISCO BCOS:pr-638 [61], data synchronization exceptions resulted in inconsistencies in block hashes. The rest TPBs stem from conflict analysis. Some parallelism mechanisms (FISCO BCOS, Solana) conduct conflict analysis upon receiving transactions. This process is complex and prone to TPBs. For example, FISCO BCOS:pr-2344 [32], FISCO BCOS:pr-2386 [30], and Solana:issue-25991 [26] are all due to the inability to identify conflicting transactions and schedule them appropriately. Besides, some TPBs require other special triggering conditions like consensus failure, resource misallocation, and incorrect configuration.

V. Design

Fig. 2 shows the framework of *Chord*. *Chord* incorporates novel designs for triggering and capturing TPBs. *Chord* first proposes a unified conflict transaction model. It designs a template contract, which contains various types of shared resources as conflict variables, and their access operations. Then *Chord* applies the conflict constructor to generate transactions with various conflict relations. It also employs a revert injector to trigger TPB-prone exception handling situations during transaction execution. *Chord* proposes two dedicated oracles for TPBs: a local-remote differential oracle and a TPS oracle, which address the non-determinism introduced by parallelism and effectively capture TPBs.

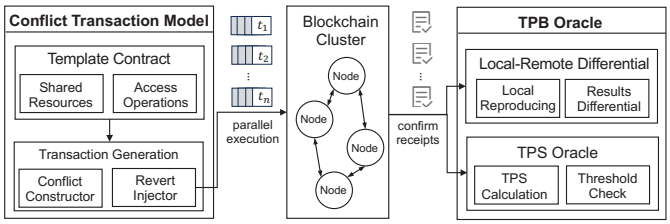


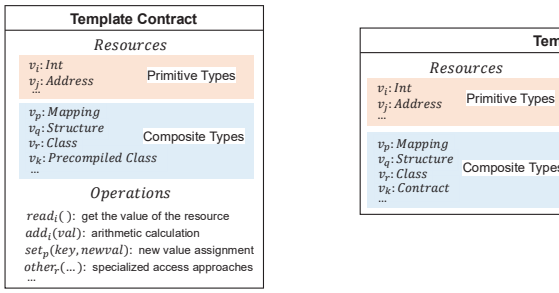
Fig. 2: The overall framework of *Chord*. *Chord* proposes the unified conflict transaction model to trigger TPBs, and establishes two dedicated oracles to detect TPBs.

A. Unified Conflict Transaction Model

To trigger TPBs, *Chord* designs the unified conflict transaction model. Building upon our findings in section IV-C, we design the template contract incorporating various conflict resources and their corresponding accessing operations. Leveraging this template contract, we generate various composite functions to create high-quality conflict transactions and error-prone conflict handling scenarios, effectively triggering TPBs.

1) Template Contract: In blockchain, transactions trigger the execution of smart contracts to interact with the blockchain system. Therefore, a template contract is first required. The contract serves as the interface for transactions to interact with the underlying blockchain transactions parallelism mechanisms. It defines a series of functions to access on-chain resources, which are available for transactions to invoke. To generate various conflicting transactions and effectively test the transaction parallelism mechanisms, the template contract should encompass a range of conflict variables and essential functions for accessing them.

Fig. 3 shows the composition of the template contract. First, it contains various shared resources, including primitive data types such as integer, address, etc. Besides, it contains composite data types that are more prone to TPBs. Specifically, *Chord* defines the mappings and arrays based on primitive data types. *Chord* also constructs more complex resources such as self-defined structures and classes. For blockchains containing pre-compiled contracts, *Chord* initiates them to create corresponding resources. Second, the template contract involves access operations for operating the shared resources, including data retrieval such as 'read', simple arithmetic calculations such as 'add', and assignments. For complex data types with



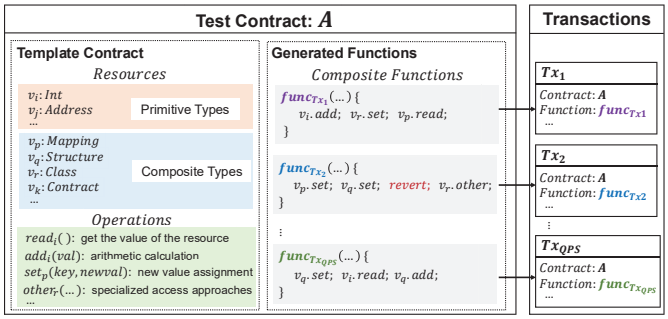


Fig. 3: The process of *Chord* generating conflict transactions. The test contract is composed of the template contract and composite functions. *Chord* generates transactions by calling the composite functions.

2) Conflict Transaction: Our findings in section IV-C indicate that blockchains are prone to TPB when processing conflict transactions and when unexpected reverts occur. Therefore, *Chord* first applies the conflict constructor to generate transactions with complex conflict relations. It then adjusts the transaction submission QPS based on the runtime

system status to increase the frequency of conflicts. *Chord* also proposes the revert injector to inject proactive reverts during execution. Therefore, *Chord* thoroughly examines the resilience of the transaction parallelism mechanisms.

Fig 3 shows the details of how *Chord* generates the test transactions. The test contract A is composed of the template contract and generated composite functions. In a test round, *Chord* instantiates the template contract to generate a number of composite functions, including $func_{Tx_1}$, $func_{Tx_2}$, ..., $func_{Tx_{QPS}}$. They are constructed using the conflict constructor and the revert injector, consisting of a series of access and revert operations. The conflict constructor distributes various access operations that access the same shared variables into different composite functions, thereby creating conflict relationships. For example, The $v_q.set$ in $func_{Tx_2}$ and $func_{Tx_{QPS}}$ are potentially conflict. Besides, the revert injector randomly inserts proactive revert operations among the access operations to trigger the exceptions during execution. Specifically, when a transaction calls $func_{Tx_2}$ and reaches the $revert$ operation, its execution will directly stop and revert, triggering the exception handling process of the transaction parallelism mechanisms. The number of the access operations within the composite function depends on the gas limit of the targeted blockchains, and can be customized. Each transaction calls one of these composite functions. For example, Tx_1 calls $func_{Tx_1}$ in test contract A, thus it will executes the access operations defined in $func_{Tx_1}$.

Chord generates a bunch of transactions for a test round, and controls the timing and frequency of transaction submission dynamically. The Query Per Second (QPS) of *Chord* is initially based on the actual Transactions Per Second (TPS) of each target blockchain. Subsequently, *Chord* gradually increases the QPS until the system's TPS stabilizes. This QPS value deliberately triggers the blockchain's transaction parallelism mechanisms, maximizing the probability of triggering conflicts, thus making it more likely to expose TPBs.

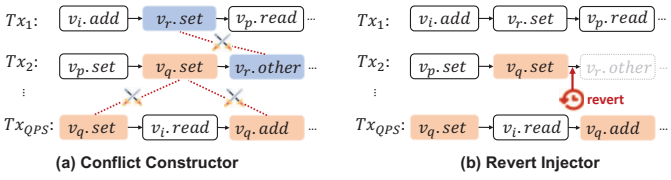


Fig. 4: *Chord* applies the conflict constructor to create complex conflict relations among transactions, and applies revert injector to trigger error-prone exception handling scenarios.

Conflict Constructor. TPBs usually occur when the transaction parallelism mechanisms deal with conflict situations. Therefore, *Chord* establishes the conflict constructor to generate various forms of conflict situations in real production. Fig. 4(a) shows the conflict relations of the transactions generated in Fig. 3. Tx_1 sets the value of v_r, a self-defined class object. Meanwhile, Tx_2 invokes the internal functions defined in the class, which also modifies v_r. Therefore, Tx_1 and Tx_2 conflict with each other, requiring the blockchain to identify the conflicts and handle them appropriately. Similarly, the set operation of v_q in Tx_2 also conflicts with the set and add operations of v_q in Tx_{QPS}.

Revert Injector. The findings in IV-C show that TPB usually arises when the transaction execution faces exceptions such as unexpected reverts, which affect the scheduling of conflicting transactions. Therefore, *Chord* introduces a revert injector to test the robustness of the parallelism mechanisms under these exceptions. Fig. 4(b) demonstrates the effect of revert injector. At the point that Tx_2 finishes $v_q.set$, *Chord* deliberately interrupts the execution by injecting proactive reverts, resulting in the rollback of Tx_2. This operation disrupts the current conflict relations, resolving the conflict between Tx_1 and Tx_2. Meanwhile, the conflict between Tx_2 and Tx_{QPS} still exists. The revert injector modifies the conflict relationships among the submitted transactions, prompting the re-scheduling and error-handling scenarios. Therefore, *Chord* can effectively test whether the transaction parallelism mechanisms can correctly manage this situation.

B. TPB Oracle

According to our finding in section IV-B, TPB often manifests without obvious phenomena, making it challenging to capture them. Besides, due to the non-determinism introduced by parallelism, existing differential oracles cannot be used for TPBs. Therefore, *Chord* proposes two dedicated oracles: a local-remote differential oracle to check the execution results, and a TPS oracle to identify liveness TPBs.

1) Local-Remote Differential Oracle: *Chord* establishes the local-remote differential oracle to validate the correctness of the calculation results. The transaction parallelism introduces uncertain execution order. Additionally, some blockchains resolve conflicts by directly reverting the execution of certain transactions. Therefore, the final result cannot be determined solely based on the submitted transactions. The differential oracles of EVMFuzzer and Fluffy focus on inconsistencies among EVMs or Ethereum clients, but TPBs can produce incorrect yet consistent results. To address this issue, *Chord* utilizes the transaction receipts after finishing the transaction confirmation to determine the actual results. Specifically, *Chord* acquires the timestamps of confirmed transactions to ascertain the exact execution sequence. Then *Chord* sequentially reproduces these transactions locally, compares the local results with on-chain results, and identifies TPBs.

Fig. 5 shows how the local-remote differential oracle verifies the correctness of the results. **Step 1**: Transactions are generated and submitted for execution. Meanwhile, *Chord* records their transaction hashes; **Step 2**: After execution, *Chord* queries the transaction receipts using the recorded hashes. The receipts contains the status of confirmation, the block hash, and the index of this transaction within the block. **Step 3**: Based on the receipts, *Chord* accesses the blocks where the transactions were included. By indexing, *Chord* determines the position of the transactions within the block, obtaining the timestamp of confirmation. **Step 4**: Leveraging these timestamps, *Chord* establishes the actual execution order

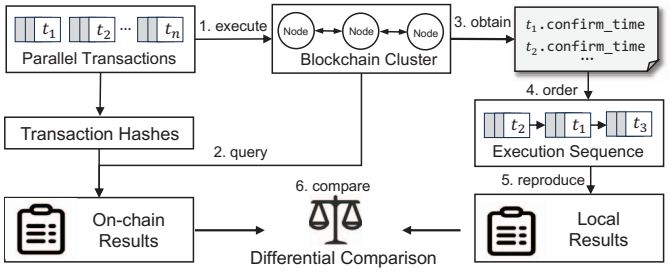


Fig. 5: The local-remote differential oracle. *Chord* obtains the transaction execution sequence and reproduces the result locally for differential checking.

of transactions. **Step 5**: *Chord* sequentially executes them locally to reproduce the expected calculation results. **Step 6**: Finally, *Chord* compares the expected results with the on-chain actual results. The differences indicate the occurrence of TPBs.

2) TPS Oracle: TPS (Transactions Per Second) is a metric for measuring the throughput of a blockchain. An abnormal decrease in TPS signals that the blockchain has lost its normal ability to process transactions. According to our study in section IV-B, TPB can manifest as an abnormal decrease in TPS. Therefore, *Chord* proposes a TPS oracle to continuously check the TPS in real time. Therefore, *Chord* effectively captures TPBs that harm the performance of the blockchain.

Unlike traditional sequential execution, some parallelism mechanisms forcely discard certain transactions to avoid conflicts when parallelizing the execution. Therefore, not all transactions will be confirmed successfully. Existing methods [21], [46] for calculating TPS require the confirmation time for all the submitted transactions, and cannot be applied to *Chord*. Therefore, *Chord* proposes the definition of TPS under the parallelism execution scenarios as follows:

$$TPS = \frac{num}{\max_{i=1}^{num}\{txs_i.ft\} - \min_{i=1}^{num}\{txs_i.st\}} \quad (1)$$

Chord calculates the blockchain's TPS after finishing the current batch of transactions. The numerator *num* represents the total number of transactions that are submitted, including both successful and failed transactions. In the denominator, $txs_i.ft$ represents the finish time of transaction i. If txs_i is successfully confirmed, the $txs_i.ft$ refers to its confirmation. If txs_i is failed, $txs_i.ft$ refers to the timestamp in its receipts. Therefore, $\max_{i=1}^{num}\{txs_i.ft\}$ refers to the latest finished transaction. $txs_i.st$ refers to the submission time of txs_i. Since that *Chord* needs to control the QPS, the test transactions are not submitted at the same time. Therefore, we use $\min_{i=1}^{num}\{txs_i.st\}$ to decide the earliest submission time.

In real-world production, fluctuations in TPS within a certain intensity and duration may be recoverable. When some transactions are not finished properly, it is hard to determine whether there is an error or a normal delay. Therefore, to avoid false positives and false negatives, *Chord* establishes a threshold for the TPS decrease ratio to identify TPBs. *Chord* records the initial TPS value at the testing QPS as the baseline. During the subsequent testing under the same QPS, if the TPS decreases by over the threshold compared to the baseline and remains unrecoverable, *Chord* flags it as a TPB. In our production environment, the threshold is set to 70% for FISCO BCOS, 50% for Sei, 70% for Solana, and 30% for Aptos. How *Chord* finds the optimal threshold for each blockchain will be discussed in section VII-A in detail.

C. Implementation

We implement *Chord* on 4 widely-used commercial blockchains with transaction parallelism mechanisms, including FISCO BCOS (version 3.5.0 [16]), Sei (version 4.1.7[55]), Solana (version 1.17.13 [58]), and Aptos (version 2.1.0 [2]). FISCO BCOS is developed in C++, supporting Solidity for programming smart contracts. Sei is implemented in Go, supporting Rust and Solidity for smart contracts. Solana and Aptos are both implemented in Rust. Solana supports Rust, C, and C++ for developing smart contracts, while Aptos uses Move. The implementation shows that *Chord* is a cross-language and cross-platform tool.

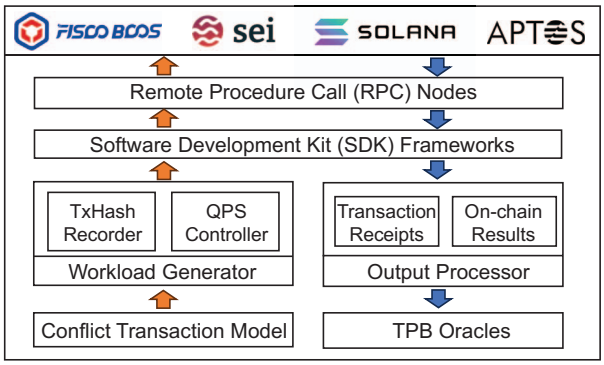


Fig. 6: The implementation of *Chord*. *Chord* interacts with the blockchain through RPC nodes and SDK. The TxHash Recorder records the transaction hashes, and the QPS controller controls the timing of submission. The hashes are used for querying receipts and on-chain results for TPB oracles.

Figure 6 outlines the implementation of *Chord*. For each targeted blockchain, *Chord* sets up a 4-node local cluster. To enhance scalability and streamline development, *Chord* encapsulates RPC operations within an SDK framework, offering well-defined interfaces for interaction. *Chord* first creates the template contracts and generates transactions based on the unified conflict transactions model. A TxHash recorder logs transaction hashes for later receipt retrieval, while a QPS controller manages the timing and rate of transaction submissions. Transactions are submitted through the SDK, triggering parallel scheduling and execution. After the execution, *Chord* utilizes an output processor to retrieve the execution results for oracle verification. Leveraging recorded transaction hashes, *Chord* fetches transaction receipts and on-chain calculation results via SDK queries. They are then used for local-remote differential oracle and TPS oracle.

VI. Evaluation

We evaluate *Chord* on four widely-used blockchains with transaction parallelism mechanisms including FISCO BCOS [14], Sei [53], Solana [57] and Aptos [38]. In section VI-B, we study the TPBs found by *Chord* to show *Chord*'s capability in detecting real-world TPBs. In section VI-C, we assess the TPS and latency of *Chord* to show that the unified conflict transaction model contributes to triggering TPBs by generating various conflict scenarios. In section VI-D, we compare *Chord* with the existing testing methods to show that *Chord* outperforms them in detecting TPBs. We address the following research questions:

- **RQ1:** Is *Chord* effective in detecting TPBs in real-world commercial blockchain systems?
- **RQ2:** How does the unified conflict transaction model of *Chord* contributes to effectively triggering TPBs?
- **RQ3:** How does *Chord* outperform the existing testing methods in detecting TPBs?

A. Evaluation Setup

Environment. We set up a 4 node local cluster for each blockchain. For detecting new TPBs, we ran FISCO BCOS version 3.5.0 [16], Sei version 4.1.7 [55], Solana version 1.17.13 [58], and Aptos version 2.1.0 [2]. For reproducing historical TPBs, we ran the corresponding versions. We conduct our experiments on an Ubuntu 22.04.3 system with the Linux kernel version 5.15.0, running on a 64-bit physical machine. The machine is equipped with 256 CPU cores (AMD EPYC 7763 64-Core Processor) and 1 TB of memory.

Metrics. We propose three metrics for evaluation. The first is the number of TPBs detected, including both historical and newly identified ones. The second is TPS, a standard metric for blockchain throughput. In parallel execution, lower TPS indicates reduced parallelization, suggesting conflicts that trigger conflict resolution. Conflicting transactions, which cannot run in parallel, require extra analysis and scheduling, as discussed in section IV-C. Since directly measuring conflicts is challenging, we use TPS to assess the quality of conflicting transactions. The third metric is latency, the time from transaction initiation to confirmation. Like TPS, latency is a key metric for user-perceived performance, providing a full picture of the number and complexity of conflicting transactions.

B. TPBs found by Chord

We evaluate the TPBs found by *Chord* to show its effectiveness in detecting TPBs. Table. I shows the TPBs that *Chord* detected within 24 hour execution. In general, *Chord* is highly effective in detecting TPBs. *Chord* successfully detects 54 TPBs across four targeted blockchains, including 10 previously unknown ones (5 in FISCO BCOS, 4 in Sei, and 1 in Aptos). We have reported the TPBs to developers and received their responses. Among the 10 newly detected TPBs, 6 are fixed. 1 is confirmed. 3 are still under processing. 3 are assigned with CVE IDs (CVE-2023-51936, CVE-2023-51937, CVE-2024-31704).

TABLE I: The 54 TPBs detected by *Chord*.

Targets	Total	Required Conflict Transaction Model?		Required TPB Oracle?	
		Yes	No	Local-Remote Differential	TPS Oracle
FISCO BCOS	26	24	2	16	10
Sei	19	17	2	11	8
Solana	7	4	3	3	4
Aptos	2	2	0	2	0

Information of Detected TPBs. As shown in Table I, *Chord* successfully detects 54 TPBs in total, including 26 in FISCO BCOS, 19 in Sei, 7 in Solana, and 2 in Aptos. 87.0% of them require *Chord*'s unified conflict transaction model to be triggered. These TPBs manifest during the conflict handling process, requiring the execution of conflicting transactions. The rest 13.0% can be triggered under a large number of parallel transactions. Besides, all TPBs require *Chord*'s TPB oracles to be identified. Specifically, 59.3% TPBs result in incorrect execution results, which require the local-remote differential oracle to be detected. 40.7% of them lead to the abnormal reduction in TPS, which require the TPS oracle to be identified. Among all the studied 50 historical TPBs, 6 cannot be detected by *Chord*. 3 of them (FISCO BCOS:pr-3047 [31], FISCO BCOS:pr-4227 [35], FISCO BCOS:pr-3036 [33]) manifest as incorrect return values or system logs. Other 2 involve insufficient parallelization (FISCO BCOS:pr-755 [29]) and system-specific side effects (Solana:pr-25562 [6]). The phenomena of them are system-specific, and are currently not covered by the general oracles of *Chord*. The rest 1 requires byzantine behavior from malicious nodes (Solana:issue-7531 [52]) to trigger. *Chord* conducts the transaction submission from the client side, thus cannot trigger this special case.

The Precision and Recall. We assess the precision and recall of *Chord* on the 50 historical TPBs. According to the information of detected TPBs, within a 24 hour testing, *Chord* reports 44 true positives and 6 false negatives. The false positives are closely related to the threshold configuration and the detailed discussion is in section VII-A. When setting the threshold to 70% for FISCO BCOS, 50% for Sei, 70% for Solana, and 30% for Aptos, *Chord* reports no false positives. Therefore, *Chord* achieves the 88% precision and 100% recall.

Severity of Detected TPBs. The TPBs detected by *Chord* have serious consequences. Among them, 32 TPBs affect execution correctness, threatening on-chain asset security: 16 cause insecure access to critical storage, leading to incorrect balances and lost transaction records; 8 result in balance errors during rent collection or asset transfers, causing direct asset losses; 4 lead to gas fee miscalculations, causing abnormal failures or extra costs; and 4 disrupt blockchain consistency, including node-state mismatches and user-view inconsistencies, potentially breaking consensus and causing transaction failures or hard forks.

Additionally, *Chord* identifies 20 TPBs that degrade system performance and liveness, preventing transaction processing. Specifically, 9 cause memory leaks or resource exhaustion,

risking system collapse; 6 result in deadlocks, leaving transactions indefinitely pending; and 5 cause transaction failures or rejection of new transactions. These issues threaten both user assets and system security.

Case Study. We present two cases demonstrating how *Chord* detects TPBs. The first is a newly discovered TPB in FISCO BCOS, assigned CVE-2023-51936. This TPB caused abnormal memory leaks and eventual node crashes. Introduced before January 2022, it remained undetected due to its reliance on complex conditions and a specialized oracle. *Chord* identified this bug using its unified conflict transaction model and TPS oracle. In the template contract, *Chord* incorporates an external contract as an access object and provides interfaces for external contract invocation. The external contract then implements a callback to the template contract, creating a scenario where the two contracts invoke each other. The DMC scheduler, which manages conflicting transactions by pausing some to avoid conflicts, failed to handle this complex inter-invocation scenario correctly. Parallel transactions created by *Chord* triggered repeated memory allocation without release, causing abnormal TPS drops. As conflicting transactions persisted, the issue escalated, ultimately leading to node crashes.

The second case is a TPB in Sei, newly discovered by *Chord* and assigned CVE-2024-31704. This TPB caused incorrect execution of on-chain resources, threatening asset security. *Chord* detected it using the unified conflict transaction model and the local-remote differential oracle. The template contract includes a resource mapping ‘address‘ to ‘int‘, simulating a balance storage variable, along with access operations such as addition, subtraction, and assignment. *Chord* used its conflict constructor to generate transactions with conflicting accesses to this resource. It also employed the revert injector to randomly revert transactions, altering conflict conditions. Sei's optimistic parallelization failed under these conditions, leading to a TPB. Within 24 hours, *Chord*'s local-remote differential oracle identified inconsistencies between the on-chain mapping state and the expected results. This TPB represents a significant threat to user asset security.

C. TPS and Latency of Chord

We evaluate the TPS and latency of *Chord* to show its effectiveness in triggering TPBs. Lower TPS and higher latency indicate more complex conflict handling scenarios. Based on our prior findings, conflict scenarios are more prone to triggering TPBs. Therefore, this metric effectively reveals the tools' capability to trigger TPBs. We propose *Chord*$^-$, a version of *Chord* without the unified conflict transaction model. *Chord*$^-$ directly applies the real-world contracts for transaction generation, without the conflict constructor and the revert injector. We compare the TPS and latency of *Chord* and *Chord*$^-$ under the same QPS to show how the unified conflict transaction model in *Chord* contributes to the triggering of TPBs. We set the QPS values based on the actual TPS of each test blockchain on our machine. We cover ranges of QPS that are both lower and higher than the actual TPS to ensure the validity of our evaluation and show the trend of variation.

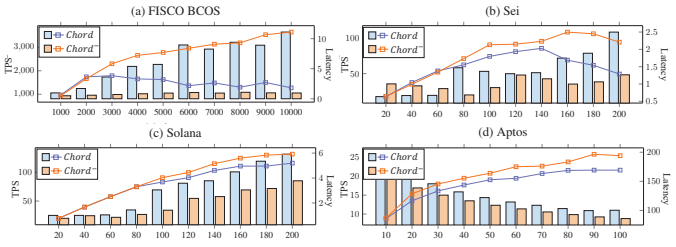


Fig. 7: The TPS and latency of *Chord* and *Chord*$^-$ as QPS increases. The lines represent TPS and the bars represent latency. *Chord* decreases the TPS by 65.13%, 47.28%, 11.85% and 15.05%, and increases the latency by 1057.27%, 97.27%, 56.36% and 16.61%.

Fig. 7 shows the trend of TPS and latency of *Chord* and *Chord*$^-$ as the QPS increases. The blue line and orange line show the trend of TPS under *Chord* and *Chord*$^-$ respectively. The blue bar refers to the latency under *Chord*, while the orange bar shows the latency under *Chord*$^-$. For each QPS value, we run the targets for 1 hour and take the average value. Compared with *Chord*$^-$, *Chord* achieves lower TPS and higher latency across all blockchains. Under the highest QPS where the TPS and latency stabilize, *Chord* achieves 65.13%, 47.28%, 11.85% and 15.05% lower TPS compared with *Chord*$^-$. Meanwhile, *Chord* increases the latency by 1057.27%, 97.27%, 56.36% and 16.61% on four blockchains respectively. With the unified conflict transaction model, *Chord* generates more conflicting transactions and triggers the conflict handling. The template contract aggregates various resources and access operations to generate conflict scenarios, improving *Chord*'s effectiveness to trigger TPBs. In contrast, using real-world contracts directly can limit conflict patterns and produce many non-conflict transactions, which reduces test effectiveness. Therefore, the unified conflict transaction model significantly contributes to triggering TPBs.

D. Comparison with existing methods

In this subsection, we compare *Chord* with existing testing methods in terms of the number of TPBs they detect and the TPS and latency.

Comparison on TPB detection. Existing testing methods cannot effectively detect TPBs. First, without specific oracles, they are unable to capture TPBs. Previous differential oracles of EVMFuzzer and Fluffy focus on inconsistencies among EVMs or Ethereum clients, but TPBs can produce incorrect yet consistent results. For example, in Solana:pr-25774, clients reach consistency on the incorrect results, leading to double-spending. Existing works cannot identify such TPBs. Besides, existing tools such as EVMFuzzer [17], EVMLab [10] and Fluffy [67] are tightly coupled with the Ethereum virtual machine, and do not support parallel transaction submission, thus cannot be applied to recently emerged blockchains that exhibit transaction parallelism mechanisms. To compare *Chord*'s TPB detection ability with them, we re-

implemented their transaction generation strategies to adapt them for our target blockchains and equip them with *Chord*'s oracles. Results show that they can only detect 7 TPBs, with 2 in FISCO BCOS, 2 in Sei, 3 in Solana, and 0 in Aptos. Lacking the ability to trigger various conflicting handling scenarios, they cannot detect most of the TPBs. In comparison, *Chord* successfully detects all the TPBs that existing methods detected, and it detects 47 more TPBs, with 26 in FISCO BCOS, 19 in Sei, 7 in Solana, and 2 in Aptos. This reveals that *Chord* outperforms the existing methods in detecting TPBs.

Comparison on TPS and latency. To evaluate the effectiveness of *Chord* in triggering TPBs, we compare the TPS and latency of *Chord* with the test suites provided by each blockchain for testing their transaction processing mechanisms. For FISCO BCOS, we compare *Chord* with its official stress testing framework [15], which can trigger its parallelism mechanisms. For Sei, Solana and Aptos, we employ their provided test contracts for transaction generation, and manually customize the parallel transaction submission framework to trigger their parallelism mechanisms. The evaluation setup is similar to section VI-C.

The results show that the TPS under *Chord* is lowered by 93.66% for FISCO BCOS, 63.16% for Sei, 27.27% for Solana, and 14.80% for Aptos compared with the existing methods. Meanwhile, *Chord* increases the latency by 1205.59% for FISCO BCOS, 219.02% for Sei, 115.34% for Solana, and 12.44% for Aptos. This reveals that *Chord* effectively triggers more conflict handling scenarios for all the blockchains. According to our finding in section IV-C, *Chord* is more effective in triggering TPBs than existing methods.

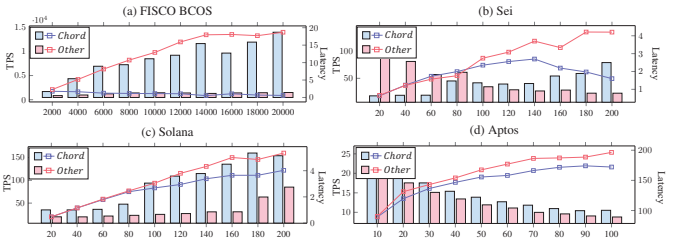


Fig. 8: The TPS and latency of *Chord* and existing methods as QPS increases. *Chord* decreases the TPS by 93.66%, 63.16%, 27.27%, and 14.80%, and increases the latency by 1205.59%, 219.02%, 115.34%, and 12.44%.

Fig. 8 shows the trend of TPS and latency of *Chord* and the existing methods as the QPS increases. The blue line and red line show the trend of TPS under *Chord* and the existing method respectively. The blue bar refers to the latency under *Chord*, while the red bar shows the latency under the existing method. For each QPS value, we run the targets for 1 hour and take the average value. When the QPS is low, the differences in TPS and latency between *Chord* and the existing methods are relatively small. Lower QPS indicates fewer parallel transactions, smaller load pressure, and fewer conflicts. Therefore, the blockchain can process the transactions faster. As the QPS increases, the TPS of *Chord* significantly falls below that of the existing method. Meanwhile, the latency of *Chord* becomes noticeably higher than the latency of the existing method. This reveals that with more parallel transactions, *Chord* creates various conflicts among them. This triggers the blockchain to handle conflicts, thereby leading to a decrease in its performance.

In conclusion, compared with existing testing methods, *Chord* successfully detects 47 more TPBs. Under the same QPS, *Chord* achieves a lower TPS and higher latency. Therefore, *Chord* can more effectively generate various conflict handling scenarios, deeply inspect the transaction parallelism mechanism, and construct triggering conditions for TPBs.

VII. Discussion

A. The Threshold of TPS Oracle

The TPS decrease threshold configuration of *Chord*'s TPS oracle impacts its identification accuracy. In blockchain systems, fluctuations in TPS and latency are common, and can be influenced by the system's resources and network conditions. A too strict threshold may result in many false positives. While a too loose threshold may impact the effectiveness of detection, causing the TPS oracle to miss some TPBs.

TABLE II: The false positives and true positives of *Chord* under various TPS oracle threshold configuration.

Threshold	FISCO BCOS		Sei		Solana		Aptos	
	FP	TP	FP	TP	FP	TP	FP	TP
10	601	10	224	8	1663	4	522	0
20	601	10	181	8	54	4	35	0
30	321	10	121	8	14	4	0	0
40	297	10	34	8	14	4	0	0
50	247	10	0	8	13	4	0	0
60	6	10	0	8	11	4	0	0
70	0	10	0	8	0	4	0	0
80	0	9	0	8	0	4	0	0
90	0	6	0	7	0	4	0	0

We collect the TPBs reported by TPS oracle within 24 hours of execution on FISCO BCOS, Sei, Solana and Aptos. Table II shows the false positives and true positives under various configurations of the threshold. A 10% threshold causes many false positives across all chains, but increasing the threshold reduces them. By setting the threshold to 70% for FISCO BCOS, 50% for Sei, 70% for Solana, and 30% for Aptos, *Chord* reports no false positives. Additionally, when the threshold is set to less than 70%, 80%, 90%, and 100% respectively, these chains can reveal all TPBs with no false negatives. Therefore, in our evaluation environment, the optimal threshold should be 70% for FISCO BCOS, 50% for Sei, 70% for Solana, and 30% for Aptos. The proper threshold configurations for different blockchains vary due to differences in consensus mechanisms, network architecture, block size, etc. They also change based on production environments, such as cluster scales and network delays.

B. Scalability on Bug Types.

Currently, *Chord* has proposed two oracles for detecting TPBs. *Chord* has been adapted to four widely used blockchains

and found 10 previously unknown bugs. However, in practice, there are still other types of bugs, such as privacy issues [69], [20], [72], which also pose threats to the security of blockchain ecosystems. For example, the leakage of critical private data, such as private key information, can seriously endanger user security and lead to irreversible economic losses. If extended with privacy oracle, by trying to access private data and checking its visibility through transactions, *Chord* can also find privacy issues. However, the privacy perspective in blockchain varies for personal and organizational data. Although privacy rules are applicable to personal data, more stringent privacy rules apply to sensitive and organizational data. The flexibility of blockchain privacy makes it hard to design a general privacy oracle for various blockchain systems. More works need to be explored to address this challenge. Privacy and other types of bugs deserve our attention in the future.

VIII. Related Work

A. Blockchain Bug Detection.

Many existing works focus on testing blockchains. SFuzz [50], ContractFuzzer [28], and V-Gas [47] use fuzzing to generate inputs and trigger hidden bugs in smart contract functions. Tools such as Securify [60], Mythril [49], and Oyente [51] model the control flow graph of smart contracts through symbolic execution to discover hidden vulnerabilities in smart contracts. LOKI [46], Tyr [5], and ByzzFuzz [62] mutate consensus messages to trigger byzantine attacks and test consensus protocols. They cannot generate transactions and cannot detect bugs in the transaction parallelism mechanisms. EVMFuzzer [17], EVMLab [10], and Fluffy [67] generate single or sequential transactions to find bugs in EVM and Ethereum consensus protocols. They cannot generate parallel conflict transactions to trigger TPBs. They also cannot adjust the QPS and trigger revert during execution, which are also essential for testing TPBs.

Nyx [71], Zhang et al. [71] and Torres et al. [59] exploits the profitable ordering of sequential transactions to detect front-running vulnerabilities. The triggering condition of front-running and TPBs share some similarity. Triggering front-running requires generating transaction sequences accessing conflict profitable variables. However, triggering TPBs has unique challenges. First, TPBs are derived from the transaction parallelism mechanisms, which require conflict and parallel transactions accessing composite data types. Additionally, triggering TPBs require unique designs during the transaction submission and execution stages. *Chord* dynamically adjusting testing QPS based on blockchain runtime TPS to trigger more conflict handling scenarios, and inserting proactive reverts to construct TPB-prone exception handling cases.

As for oracles, existing differential oracles cannot be used for TPBs. For example, EVMFuzzer and Fluffy focus on inconsistencies among EVMs or Ethereum clients, but TPBs can produce incorrect yet consistent results across different platforms. Therefore, their oracles cannot be used for capturing TPBs. In contrast, *Chord* proposes the local-remote differential oracle, using transaction confirmation receipts to reproduce the actual execution order and expected results, rather than manipulating the transaction order like front-running attacks, for differential testing. This addresses the non-determinism introduced by parallelism and effectively captures many TPBs.

B. Concurrency Bugs Detection.

Many existing works aim at detecting concurrency bugs in various systems. TaxDC [41] classifies concurrency bugs into Local Concurrency (LC) Bugs and Distributed Concurrency (DC) Bugs and discusses the triggering conditions.. CHESS [48] and TSVD [42] mine LCBugs in various concurrent programs by controlling thread scheduling or monitoring the invocation of thread-unsafe methods. These LCBugs are typically caused by concurrent read/write conflicts or the concurrent execution of the same operation. DCatch [44] and CloudRaid [45] discover DCBugs in multiple distributed cloud systems by simulating various concurrency mechanisms and message orders. Jepsen [27] explores concurrency bugs in distributed systems by injecting random network partition faults. These DCBugs are caused by the nondeterministic order of distributed events, such as the sending and receiving of messages, node crashes, restarts, and timeouts.

Although concurrency bugs and TPBs share similarities in their triggering conditions, as both are caused by conflicting access operations on shared resources, detecting TPBs presents unique challenges. Concurrency bugs are caused by locking and synchronization mechanisms, while TPBs arise from the transaction parallelism mechanisms that require conflict transactions accessing composite resources to trigger. Besides, existing works focus on conflict transaction relations in test case generation stage. But *Chord* also includes specific designs in transaction submission and execution stage to construct various TPB-prone scenarios. Specifically, during submission, *Chord* dynamically adjusts testing QPS based on blockchain runtime TPS to trigger more conflict handling scenarios. Besides, *Chord* injects proactive reverts during execution stage to construct error-prone exception handling logic.

IX. Conclusion

In this work, we conducted a thorough analysis of real-world TPBs in four blockchains to study their symptoms and root causes. Based on our findings, we propose *Chord*, aiming at detecting TPBs. *Chord* proposes a unified conflict transaction model to generate various conflicting transactions and trigger TPBs. Besides, *Chord* applies two oracles for identifying TPBs. *Chord* successfully detects 54 TPBs, including 10 previously unknown ones, showing its effectiveness in detecting TPBs. Besides, evaluations on TPS and latency reveal that *Chord* outperforms the existing methods in triggering TPBs.

X. Acknowledgement

This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000), NSFC Program (No. U2441238, 62021002, 62332004), and China Postdoctoral Science Foundation (2024M761690, GZC20240828).

REFERENCES

[1] Salem Alqahtani and Murat Demirbas. Bottlenecks in blockchain consensus protocols. In *2021 IEEE International Conference on Omni-Layer Intelligent Systems (COINS)*, pages 1–8. IEEE, 2021.

[2] Aptos. aptos-core 2.1.0. https://github.com/aptos-labs/aptos-core/tree/aptos-cli-v2.1.0, 2024.

[3] Robert P. Bartlett and Justin McCrary. How rigged are stock markets? evidence from microsecond timestamps. *Journal of Financial Markets*, 45:37–60, 2019.

[4] Andrew Brook. Low-latency distributed applications in finance. *Commun. ACM*, 58(7):42–50, jun 2015.

[5] 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 2517–2532. IEEE, 2023.

[6] ckamm. Findpacketsenderstake: Remove parallelism to improve performance. https://github.com/solana-labs/solana/pull/25562, 2022.

[7] codchen. Fix race conditions detected by race detector. https://github.com/sei-protocol/sei-chain/pull/647/commits/f846966df1601df4a9a6f5e6994a9615167773ab, 2023.

[8] Cyson. Fix race condition when optimisticprocessinginfo is not nil. https://github.com/sei-protocol/sei-chain/pull/637/commits/effa090c120ab7c3337b22e049e7393ace36bae1, 2023.

[9] Ethereum. Danksharding. https://ethereum.org/en/roadmap/danksharding/, 2020. Accessed at June 6, 2024.

[10] Ethereum. Evmlab. https://github.com/ethereum/evmlab, 2020.

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

[12] FISCO. Transaction parallelism. https://fisco-bcos-documentation.readthedocs.io/zh-cn/latest/docs/design/parallel/dag.html, 2023.

[13] Fisco. Dmc deterministic multi-contract parallel. https://fisco-bcos-doc.readthedocs.io/zh-cn/latest/docs/design/parallel/DMC.html, 2024.

[14] FISCO. Fisco bcos. http://www.fisco-bcos.org, 2024.

[15] FISCO-BCOS. java-sdk-demo. https://github.com/FISCO-BCOS/java-sdk-demo, 2022.

[16] FISCO-BCOS. Fisco-bcos release-3.5.0. https://github.com/FISCO-BCOS/FISCO-BCOS/tree/release-3.5.0, 2024.

[17] 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.

[18] Rati Gelashvili, Alexander Spiegelman, Zhuolun Xiang, George Danezis, Zekun Li, Dahlia Malkhi, Yu Xia, and Runtian Zhou. Block-stm: Scaling blockchain execution by turning ordering curse to a performance blessing, 2022.

[19] godmodegalactus. Transactions dropped in sig verify stage because of improper batching. https://github.com/solana-labs/solana/issues/29895, 2023.

[20] Ryan Henry, Amir Herzberg, and Aniket Kate. Blockchain access privacy: Challenges and directions. *IEEE Security & Privacy*, 16(4):38–45, 2018.

[21] Hyperledger. Hyperledger caliper. https://hyperledger.github.io/caliper/, 2021.

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

[23] imtypist. stress test, checking result failed. https://github.com/FISCO-BCOS/FISCO-BCOS/issues/2101, 2022.

[24] jdkuangxx. After the keypage is closed, the node is pressed, and it is found that the node memory keeps increasing and cannot be recycled. https://github.com/FISCO-BCOS/FISCO-BCOS/issues/3826, 2023.

[25] jeffwashington. collect rent from multiple partitions in parallel. https://github.com/solana-labs/solana/pull/25774, 2022.

[26] jeffwashington. parallel rent collection avoids overlapping ranges. https://github.com/solana-labs/solana/pull/25991, 2022.

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

[28] Bo Jiang, Ye Liu, and Wing Kwong Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pages 259–269, 2018.

[29] JimmyShi22. Fix parallel bugs && add unitest. https://github.com/FISCO-BCOS/FISCO-BCOS/pull/755/commits/20911f6fb7d6143c1a35a0bf3a23d05cbee97047, 2019.

[30] JimmyShi22. Fix concurrent bug. https://github.com/FISCO-BCOS/FISCO-BCOS/pull/2386/commits/333ca26b1d6bb101da3918feb7944a7633d843e8, 2022.

[31] JimmyShi22. Fix dmc callback core. https://github.com/FISCO-BCOS/FISCO-BCOS/pull/3047/commits/30bb41802f4b70a3659c4f8f7e1a450c5d8e2f25, 2022.

[32] JimmyShi22. Fix dmc transfer bug and optimize code. https://github.com/FISCO-BCOS/FISCO-BCOS/pull/2344/commits/e06529eb78cfa8c612742bb915e28a3be43fcfad, 2022.

[33] JimmyShi22. Fix executor version bug && fix dmc getcode block bug && add drop redundant switching requests logic in executor switch. https://github.com/FISCO-BCOS/FISCO-BCOS/pull/3036/commits/eebd83276897cd72f37d21df2ec48ee10c39155a, 2022.

[34] JimmyShi22. fix sharding dmc scheduler bug and optimize txpool initialize process. https://github.com/FISCO-BCOS/FISCO-BCOS/pull/3446/commits/249f71826ef31ea4f12390d2e8bb3cf8145d4a2d, 2023.

[35] JimmyShi22. fix some small bugs part 2. https://github.com/FISCO-BCOS/FISCO-BCOS/pull/4227/commits/1296467de28b76e93dd7c744d0bd532a782e0469, 2024.

[36] jstarry. Cross program instructions may need size restrictions. https://github.com/solana-labs/solana/issues/12441, 2020.

[37] kyonRay. fix kv table precompiled create parallel bug, mv precompiledcodec to bcos-codec/wrapper. https://github.com/FISCO-BCOS/FISCO-BCOS/pull/2339/commits/99446dda319c1424a70ca08ef315ede02e981618, 2022.

[38] Aptos Lab. Aptos lab. https://aptoslabs.com, 2024.

[39] LCyson. Replace bach32 with accaddress in dex module). https://github.com/sei-protocol/sei-chain/pull/483, 2023.

[40] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F Lukman, and Haryadi S Gunawi. {SAMC}:{Semantic-Aware} model checking for fast discovery of deep bugs in cloud systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 399–414, 2014.

[41] Tanakorn Leesatapornwongsa, Jeffrey F Lukman, Shan Lu, and Haryadi S Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the twenty-first international conference on architectural support for programming languages and operating systems*, pages 517–530, 2016.

[42] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 162–180, New York, NY, USA, 2019. Association for Computing Machinery.

[43] Huizhong Li, Yujie Chen, Xiang Shi, Xingqiang Bai, Nan Mo, Wenlin Li, Rui Guo, Zhang Wang, and Yi Sun. Fisco-bcos: An enterprise-grade permissioned blockchain system with high-performance. SC '23, New York, NY, USA, 2023. Association for Computing Machinery.

[44] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian. Dcatch: Automatically detecting distributed concurrency bugs in cloud systems. *ACM SIGARCH Computer Architecture News*, 45(1):677–691, 2017.

[45] Jie Lu, Feng Li, Lian Li, and Xiaobing Feng. Cloudraid: hunting concurrency bugs in the cloud via log-mining. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 3–14, New York, NY, USA, 2018. Association for Computing Machinery.

[46] 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 *NDSS*, 2023.

[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*, 23(3):1–22, 2023.

[48] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 267–280, USA, 2008. USENIX Association.

[49] mythril. Mythril is a security analysis tool for evm bytecode. https://github.com/Consensys/mythril, 2024.

[50] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 778–788, 2020.

[51] oyente. An analysis tool for smart contracts. https://github.com/enzymefinance/oyente, 2020.

[52] pgarg66. Malicious leader can flood cluster with excessive pre-processed transactions. https://github.com/solana-labs/solana/issues/7531, 2019.

[53] Sei. Sei. https://www.sei.io, 2024.

[54] Sei. Sei is the first parallelized evm. https://v2.docs.sei.io/overview, 2024.

[55] Sei. Sei v4.1.7-evm-devnet. https://github.com/sei-protocol/sei-chain/tree/v4.1.7-evm-devnet-fix, 2024.

[56] Jiri Simsa, Randy Bryant, and Garth Gibson. {dBug}: Systematic evaluation of distributed systems. In *5th International Workshop on Systems Software Verification (SSV 10)*, 2010.

[57] Solana. Powerful for developers. fast for everyone. https://solana.com, 2024.

[58] Solana. solana-cli 1.17.13. https://github.com/solana-labs/solana/tree/v1.17, 2024.

[59] Christof Ferreira Torres, Ramiro Camino, et al. Frontrunner jones and the raiders of the dark forest: An empirical study of frontrunning on the ethereum blockchain. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1343–1359, 2021.

[60] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 67–82, 2018.

[61] vita dounai. fix the bug that in parallel mode block hash is not consistent. https://github.com/FISCO-BCOS/FISCO-BCOS/pull/638/commits/4c7671f7c66655108df392a1b0a6809d94558d68, 2019.

[62] Levin N. Winter, Florena Buse, Daan de Graaf, Klaus von Gleissenthall, and Burcu Kulahcioglu Ozkan. Randomized testing of byzantine fault tolerant algorithms. *Proc. ACM Program. Lang.*, 7(OOPSLA1), apr 2023.

[63] Lu Xu, Wei Chen, Zhixu Li, Jiajie Xu, An Liu, and Lei Zhao. Locking mechanism for concurrency conflicts on hyperledger fabric. In *Web Information Systems Engineering–WISE 2019: 20th International Conference, Hong Kong, China, January 19–22, 2020, Proceedings 20*, pages 32–47. Springer, 2019.

[64] Xiaoqiong Xu, Xiaonan Wang, Zonghang Li, Hongfang Yu, Gang Sun, Sabita Maharjan, and Yan Zhang. Mitigating conflicting transactions in hyperledger fabric-permissioned blockchain for delay-sensitive iot applications. *IEEE Internet of Things Journal*, 8(13):10596–10607, 2021.

[65] Anatoly Yakovenko. Sealevel — parallel processing thousands of smart contracts. https://medium.com/solana-labs/sealevel-parallel-processing-thousands-of-smart-contracts-d814b378192, 2019.

[66] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. Modist: Transparent model checking of unmodified distributed systems. In *NSDI'09*, pages 213–228, 2009.

[67] 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.

[68] Xinhao Yuan and Junfeng Yang. Effective concurrency testing for distributed systems. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1141–1156, 2020.

[69] Rui Zhang, Rui Xue, and Ling Liu. Security and privacy on blockchain. *ACM Computing Surveys (CSUR)*, 52(3):1–34, 2019.

[70] Wuqi Zhang, Lili Wei, Shing-Chi Cheung, Yepang Liu, Shuqing Li, Lu Liu, and Michael R Lyu. Combatting front-running in smart contracts: Attack mining, benchmark construction and vulnerability detector evaluation. *IEEE Transactions on Software Engineering*, 49(6):3630–3646, 2023.

[71] Wuqi Zhang, Zhuo Zhang, Qingkai Shi, Lu Liu, Lili Wei, Yepang Liu, Xiangyu Zhang, and Shing-Chi Cheung. Nyx: Detecting exploitable front-running vulnerabilities in smart contracts. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 146–146. IEEE Computer Society, 2024.

[72] Haider Dhia Zubaydi, Pál Varga, and Sándor Molnár. Leveraging blockchain technology for ensuring security and privacy aspects in internet of things: a systematic literature review. *Sensors*, 23(2):788, 2023.