

Nexus: A Novel Transaction Processing Framework for Permissioned Blockchain

Shengjie Guan , Rongkai Zhang , Qiuyu Ding , Mingxuan Song , Zhen Xiao , *Senior Member, IEEE*,
Jieyi Long , Mingchao Wan, Taifu Yuan, and Jin Dong 

Abstract—The transaction execution layer is a key determinant of throughput in permissioned blockchains. While recent Shared Memory Pools (SMP)-based approaches improve throughput by enabling all consensus nodes to participate in transaction packaging, they face two fundamental limitations. First, the performance bottleneck shifts from the consensus layer to the transaction execution layer as transaction number confirmed in a round increases. Second, these approaches are vulnerable to “transaction duplication” attacks where malicious clients can simultaneously send the same transaction to multiple consensus nodes, thereby decreasing the number of valid transactions in block proposals. To address these limitations, this paper introduces *Nexus*, a novel blockchain transaction processing framework with high scalability. *Nexus* leverages the idle computational resources of full nodes to enable transaction execution in parallel with the consensus. Moreover, *Nexus* allows each node to handle only a fraction of the total transactions and share execution results with others. This approach reduces overall transaction execution time, increases throughput, and decreases latency. Lastly, *Nexus* introduces a transaction partitioning mechanism that effectively addresses the “transaction duplication” attack and achieves load balancing between clients and consensus nodes. Our implementation of *Nexus* demonstrates significant improvements: throughput increases by 4x to 15x, and latency is reduced by 50% to 70%.

Index Terms—Latency, permissioned blockchain, throughput, transaction processing.

I. INTRODUCTION

PERMISSIONED blockchains are distributed ledgers typically maintained by business consortia or multiple organizations [1], [2]. At present, most permissioned blockchains adopt leader-based Byzantine Fault Tolerance (BFT) consensus protocols to achieve higher performance [3], [4], [5], [6], [7]. However, these consensus protocols suffer from the

leader bottleneck issue: the bandwidth of non-leader consensus nodes is significantly underutilized, primarily because their roles in the consensus process are limited to voting on block proposals.

To address this issue, some studies employ Erasure Coding (EC) [8], [9], [10] to distribute block proposals among consensus nodes, thereby alleviating the bandwidth burden on the leader node by involving non-leader consensus nodes in the distribution. However, during the subsequent voting phase, the bandwidth resources of all non-leader consensus nodes remain underutilized [9]. Furthermore, from a global perspective, the redundant chunk distribution introduced by EC causes the total outbound bandwidth consumed for block proposal distribution to be higher than that without EC. This reduces the bandwidth available for transaction distribution and limits the number of transactions available to the leader during block proposal generation. Consequently, the upper bound of consensus throughput is lowered.

To overcome the above shortcomings of EC, recent research efforts [11], [12], [13], [14], [15], exemplified by *Narwhal* [11] and *Predis* [12], propose using Shared Memory Pools (SMP) [16] to address the leader bottleneck. SMP-based protocols fundamentally differ from EC-based approaches in that they decouple transaction distribution from the consensus process. This design allows each consensus node to independently and concurrently generate block substructures composed of transactions and distribute them to other consensus nodes. By leveraging the otherwise idle bandwidth of consensus nodes during the voting phase, SMP-based protocols enable more transactions to be confirmed in a single consensus round, thereby significantly improving throughput. However, SMP-based protocols still face two critical limitations that hinder their ability to fully realize their performance potential. Addressing these critical limitations for SMP-based protocols constitutes the primary motivation of this paper.

Firstly, the performance evaluations of SMP-based protocols in existing research efforts consistently neglect the transaction execution phase. However, in permissioned blockchains, transaction execution is an essential phase in the transaction lifecycle [17], since consensus nodes typically proceed to the next consensus round only after executing all transactions within the current block. Therefore, although these SMP-based protocols can package a large number of transactions in a single consensus round, they concurrently require consensus nodes to execute more transactions. For nodes with limited bandwidth and computational resources [18], this leads to prolonged

Received 22 April 2025; revised 20 January 2026; accepted 21 January 2026. Date of publication 27 January 2026; date of current version 23 February 2026. This work was supported in part by the National Key R&D Program of China under Grant 2023YFB2703800 and in part by the Beijing Advanced Innovation Center for Future Blockchain and Privacy Computing. Recommended for acceptance by J. Carretero. (Corresponding authors: Zhen Xiao; Jin Dong.)

Shengjie Guan, Rongkai Zhang, Qiuyu Ding, Mingxuan Song, and Zhen Xiao are with the School of Computer Science, Peking University, Beijing 100871, China (e-mail: guanshengjie@stu.pku.edu.cn; rkzhang@stu.pku.edu.cn; dingqiuyu@stu.pku.edu.cn; songmingxuan@stu.pku.edu.cn; xiaozhen@pku.edu.cn).

Jieyi Long is with Theta Labs, San Jose, CA 95128 USA (e-mail: jieyi@thetalabs.org).

Mingchao Wan, Taifu Yuan, and Jin Dong are with the Beijing Academy of Blockchain and Edge Computing, Beijing 100190, China (e-mail: chainmaker@baec.org.cn; yuantaifu@baec.org.cn; dongjin@baec.org.cn).

Digital Object Identifier 10.1109/TPDS.2026.3658222

execution times, network congestion, and increased transaction confirmation latency, ultimately degrading the Quality of Service (QoS) for users [19]. Our experiments demonstrate that when the transaction execution phase is considered, the throughput of these SMP-based protocols drops to only 5% to 10% of their reported values, while latency increases by 2x to 3x. In essence, this optimization shifts the performance bottleneck from the consensus layer to the transaction execution layer [19], [20], [21].

Secondly, these SMP-based protocols overlook the potential “transaction duplication” attack. In their assumptions, each client is assumed to be honest and to send transactions to a unique consensus node [11], [12], [13], [14], [15], [22]. However, this assumption does not hold in practical environments [23], [24], [25]. A malicious client can send the same transaction to multiple consensus nodes. Such an attack can significantly reduce the number of valid transactions in block proposals. In the worst cases, the throughput of these SMP-based protocols can drop to $\frac{1}{N_c}$, where N_c is the number of consensus nodes, even falling below the performance of PBFT [3] and HotStuff [4].

To address these limitations, we further investigate SMP-based protocols and discover that block substructures, such as bundles [12] and microblocks [13], generated by consensus nodes essentially represent a pre-ordering of a batch of transactions. In SMP-based protocols, the leader node only orders block substructures when generating block proposals. This process does not modify the order of transactions within each block substructure. Additionally, in permissioned blockchains, full nodes¹ remain idle while awaiting consensus results from consensus nodes, leading to a significant waste of computational resources.

In light of the aforementioned findings, this paper proposes *Nexus*, a novel transaction processing framework for permissioned blockchains adopting SMP-based protocols. *Nexus* decouples transaction execution from consensus and leverages the otherwise idle computational resources of full nodes. Specifically, *Nexus* allows consensus nodes to forward the block substructures they generate to the full nodes affiliated with them. These full nodes can then pre-execute the transactions within block substructures in parallel while consensus nodes are engaged in the consensus process. Additionally, *Nexus* allows consensus nodes to collect pre-execution results from full nodes and forward them to other consensus nodes. Therefore, each node handles only $\frac{1}{N_c}$ of the total transactions, thereby reducing the number of transactions that each node needs to execute. Once the consensus result is received, both full nodes and consensus nodes can use the pre-execution results to accelerate the transaction execution phase. Through this design, *Nexus* addresses the transaction execution bottleneck in SMP-based protocols, enhancing throughput while reducing transaction latency.

Furthermore, *Nexus* implements an efficient transaction partitioning mechanism. This mechanism deterministically maps each client to a unique consensus node, ensuring that each transaction is handled by only one consensus node, effectively

preventing “transaction duplication” attacks. Additionally, this mechanism achieves load balancing between clients and consensus nodes. Compared to the load balancing strategy adopted by *Stratus* [13], *Nexus* does not require frequent communication among consensus nodes, thereby eliminating the risk of load transfer failures and additional bandwidth consumption. Moreover, this transaction partitioning mechanism can effectively resist both hotspot and censorship attacks. When consensus nodes dynamically join or leave, *Nexus* can still maintain load balancing, demonstrating better robustness and scalability. Therefore, *Nexus* achieves a harmonious integration of attacks defense and load balancing.

In summary, this paper makes the following contributions:

- We propose *Nexus*, a novel transaction processing framework for permissioned blockchains adopting SMP-based protocols. *Nexus* decouples transaction execution from consensus and leverages the idle computational resources of full nodes to execute transactions in parallel with consensus. Additionally, *Nexus* also reduces the number of transactions each node needs to handle.
- We propose a comprehensive transaction partitioning mechanism in *Nexus* based on consistent hashing that effectively addresses the “transaction duplication” attack against SMP-based protocols and achieves load balancing between clients and consensus nodes.
- We implement *Nexus* based on HotStuff and comprehensively evaluate its performance. Experimental results show that *Nexus* significantly improves throughput, achieving 4x to 15x the baseline and reducing latency by 50% to 70%, while maintaining superior performance even under malicious node attacks.

II. SYSTEM MODEL

Nexus works within permissioned blockchains. Hence, the assumptions in this paper are as follows:

Nodes in a permissioned blockchain are classified into three main types: clients, full nodes, and consensus nodes. Clients sign and generate transactions and submit them to a full node. Full nodes synchronize new blocks from consensus nodes to maintain the distributed ledger and provide services to clients. Consensus nodes, selected from full nodes, package transactions from other full nodes into blocks and extend the distributed ledger through a leader-based BFT consensus protocol. In *Nexus*, it assumes that there are N full nodes, among which at most f nodes may be malicious. Based on this, the number of consensus nodes in *Nexus* satisfies $N_c \geq 3f + 1 (N_c \leq N)$. The information of consensus nodes and their affiliated full nodes is periodically maintained through consensus. Therefore, clients can obtain their latest information from any node.

Nexus is built upon a series of SMP-based protocols [11], [12], [13], [14], [15] that are designed for leader-based BFT consensus. Consequently, *Nexus* inherits both the Byzantine threat model and the communication model [3], [4]. For example, malicious nodes may produce arbitrary values, delay or omit messages, and collude with one another, but cannot forge the signatures of honest nodes. Furthermore, *Nexus* is presumed to be based on a partially synchronous network model, meaning

¹Full nodes, except for those selected as consensus nodes, do not participate in consensus processes, but still need to synchronize new blocks from consensus nodes to maintain the distributed ledger.

that after a Global Stabilization Time (GST) event, messages will arrive within a known time Δ . Although the GST is guaranteed to occur within a finite time, its exact timing is unknown [26].

III. NEXUS DESIGN

Nexus is a novel transaction processing framework designed for permissioned blockchains that adopt SMP-based protocols. It consists of three main components: **Transaction Partitioning Mechanism**, which tackles the “transaction duplication” attack; **Pre-Execution**, which utilizes the idle computational resources of full nodes to pre-execute transactions within block substructures; and **Feedback on Execution**, which shares the pre-execution results of block substructures between full nodes and consensus nodes. This section first introduces the three components of *Nexus* individually and then provides a detailed explanation of the overall *Nexus* workflow.

A. Transaction Partitioning Mechanism

In SMP-based protocols such as *Narwhal* [11] and *Predis* [12], consensus nodes independently and continuously package transactions to generate block substructures, without any interaction between their respective transaction pools. The essence of this design is the mandatory transaction pool synchronization. However, this design introduces a vulnerability for malicious clients or nodes providing transactions to consensus nodes. Specifically, malicious nodes may exploit this by sending the same transaction to all consensus nodes. Consequently, the same transaction appears in the transaction pools of all consensus nodes, and is packaged into block substructures. These substructures are then distributed to other consensus nodes. When the leader node receives block substructures from other consensus nodes, these block substructures contain duplicate transactions that already exist in its local transaction pool. In the worst-case scenario where all clients send their transactions to all consensus nodes, only $\frac{1}{N_c}$ of the transactions in all block substructures would be valid. Therefore, when facing the “transaction duplication” attack, the transactions per second (TPS) of SMP-based protocols drops to $\frac{1}{N_c}$ of the TPS under normal conditions.

To demonstrate the severity of this attack, *Predis* [12], an SMP-based protocol with the best performance, is implemented based on Hotstuff [27]. The maximum TPS without transaction execution phase is compared among the original HotStuff (Basic-HotStuff), *Predis*, and *Predis* under the “transaction duplication” attack (*Predis*-attacked). The number of consensus nodes is set to 4, 8, and 16. The inter-node RTT is set to 30 ms, and each node has a bandwidth of 100 Mb/s. In the attack scenario, all clients send their transactions to all consensus nodes. The experimental results are presented in Fig. 1. As depicted in Fig. 1, *Predis*-attacked exhibits significant TPS degradation compared to *Predis*, with its TPS dropping even below that of Basic-HotStuff. Furthermore, the performance degradation becomes more severe as the number of consensus nodes increases. These results highlight the “transaction duplication” attack as a critical challenge that SMP-based protocols must address effectively.

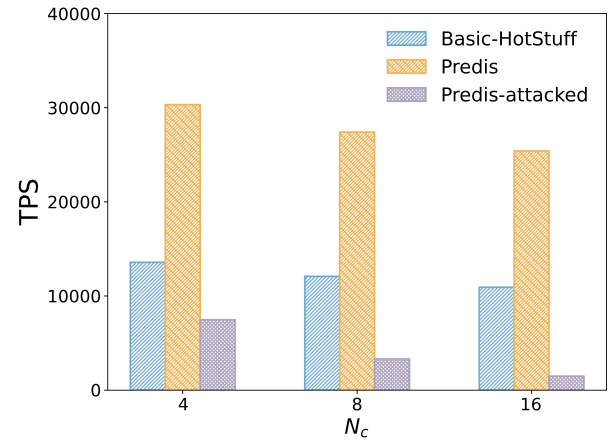


Fig. 1. “Transaction Duplication” attack with 4, 8, 16 nodes.

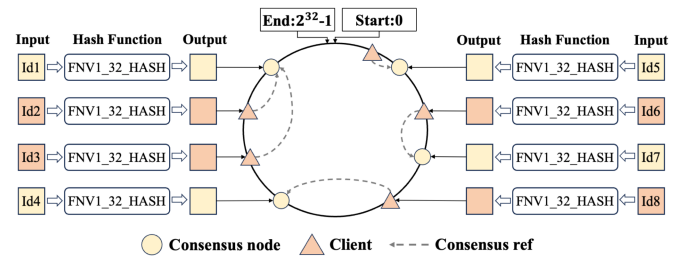


Fig. 2. Mapping clients and consensus nodes on the hash ring.

This paper proposes a comprehensive transaction partitioning mechanism based on consistent hashing [28] to address the “transaction duplication” attack. As shown in Fig. 2, this mechanism maps the entire address hash value space onto a virtual ring. The hash space on the ring ranges clockwise from 0 to $2^{32} - 1$. Under the assumption that both clients and consensus nodes have unique addresses, each is mapped to a unique point on the hash ring. Since the information of consensus nodes is maintained through consensus, clients and consensus nodes can obtain the same consistent hash ring. When a client generates a new transaction, it searches clockwise on the hash ring starting from the position of its own hash value. The first consensus node encountered is assigned to receive and package transactions from this client. The client then forwards the transaction to a full node affiliated with that consensus node. Each full node is affiliated with only one consensus node and forwards the transaction to it. Upon receiving a new transaction, a consensus node evaluates whether it falls within its assigned scope based on the address of the client in the transaction. If the transaction falls under its scope, the consensus node verifies its correctness and adds it to the transaction pool. If not, the transaction is discarded. Therefore, the “transaction duplication” attack implemented by malicious nodes is rendered ineffective.

Furthermore, *Nexus* adds virtual nodes for each consensus node. Through string concatenation based on consensus node addresses, *Nexus* maps a single consensus node to multiple points on the hash ring. This approach ensures a more uniform distribution of consensus nodes on the hash ring, thereby avoiding data skew issues [29] where a single consensus node

Algorithm 1: Transaction Partitioning Mechanism.

```

1: Input: Client address  $c$ , set of consensus nodes  $N_c$ ,
   transaction  $tx$ , string concatenation rules  $P$ .
2: Output: Transaction forwarded to corresponding
   consensus node.
3:  $N_{virt} \leftarrow \bigcup_{N_i \in N_c} \{N_i || P_j \mid P_j \in P\}$  {Create nodes
   set}
4: // Clients:
5:  $h_c \leftarrow H(c)$  {Hash client address}
6:  $\mathcal{R} \leftarrow \text{HashRing}(N_{virt})$  {Build hash ring}
7:  $N^* \leftarrow \text{NextOnRing}(h_c, \mathcal{R})$  {Find first node
   clockwise}
8: Send  $tx$  to a full node  $F^*$  affiliated with  $N^*$ 
9: // Failure/Timeout Handling
10: if no ack from  $F^*$  after timeout then
11:    $F^* \leftarrow \text{SelectNewFullNode}(N^*)$ 
12:   if failures from  $f + 1$  full nodes then
13:     Collect  $f+1$  signatures as proof  $\Omega$ 
14:      $N^{**} \leftarrow \text{NextOnRing}(N^*, \mathcal{R})$ 
15:     Send  $tx || \Omega$  to a full node  $F^{**}$  affiliated with  $N^{**}$ 
16:   end if
17: end if
18: // Consensus Nodes:
19: if  $h_c$  is under scope of  $N^*$  then
20:   Add  $tx$  to  $TxPool_{N^*}$ 
21: else
22:   Discard  $tx$ 
23: end if

```

becomes overloaded. To defend against hotspot and censorship attacks, the string concatenation rules in *Nexus* are periodically updated via consensus. Therefore, *Nexus* achieves effective load balancing between clients and consensus nodes while addressing “transaction duplication” attacks.

In practice, full nodes that forward transactions for clients may behave maliciously, such as delaying or not forwarding transactions, which threatens the liveness of the system. *Nexus* employs a timeout mechanism that allows clients to permanently replace full nodes responsible for transaction forwarding. Additionally, if a transaction is not packaged by a consensus node despite being relayed by $f + 1$ full nodes, this may indicate a censorship attack [10], [30]. In this case, the client can wait for the update of the string concatenation rules. After the rules are updated, the positions of consensus nodes on the hash ring will change, and the client’s transactions will be processed by other consensus nodes. Alternatively, the client can collect signatures on both the transaction and the address of the consensus node from these $f + 1$ full nodes. The client then attaches these signatures to the transaction and seeks the second node mapped to the consensus node in the clockwise direction on the hash ring to process this transaction. Upon receiving such a transaction, the corresponding consensus node processes it after verifying the authenticity of the attached signatures. The workflow of the transaction partitioning mechanism in *Nexus* is presented in Algorithm 1.

An adversary may also attempt to launch a hotspot attack by creating a large number of clients mapped to a specific consensus node. However, *Nexus* employs consistent hashing to map node addresses onto a hash ring, making node positions inherently unpredictable. Moreover, since *Nexus* operates in a permissioned blockchain, strict admission control policies limit the number of clients an adversary can register. Even if numerous clients could be registered, the strong collision resistance of the hash function makes it computationally infeasible to generate clients that target specific positions on the ring. Finally, string concatenation rules are periodically updated through consensus, which invalidates any precomputed attack vectors and further enhances resistance against long-term targeted attacks. Collectively, these defense mechanisms ensure robustness and security against hotspot attacks.

Compared to traditional transactions allocation schemes that use a fixed number of bits from the client address, *Nexus* disrupts the original address pattern through additional address mapping, achieving a more even distribution of nodes on the hash ring. Consequently, *Nexus* ensures a more balanced load between clients and consensus nodes. Moreover, the preimage resistance [31] of the hash algorithm significantly increases the difficulty of mounting hotspot attacks on consensus nodes. *Stratus* [13] proposes a load-balancing method between clients and consensus nodes. However, its approach relies on proactive detection among nodes. This detection leads to additional communication overhead, adversely affecting the throughput. Furthermore, the proactive detection method carries a risk of failure. The mechanism in *Nexus* is a passive load-balancing approach that does not require additional communication and active detection among nodes. Another advantage of the mechanism in *Nexus* lies in its resilience to node churn. When consensus nodes join or leave, only the nodes corresponding to the immediately adjacent positions on the hash ring are affected, while others remain unaffected. As a result, consensus nodes joining or leaving only require the relocation of a small fraction of clients on the ring. Accordingly, the mechanism in *Nexus* offers superior fault tolerance and scalability.

B. Pre-Execution

Consensus nodes can only proceed to the next consensus round after executing all transactions in the current block. Similarly, full nodes can serve clients based on the latest state after executing all transactions in the current block. Therefore, the transaction execution time is a critical factor that affects both system throughput and transaction latency.

The predominant transaction processing workflow in permissioned blockchains is the “Distribution-Consensus-Execution” (DiCE) workflow as shown in Fig. 3. In this sequential workflow, once a node distributes a transaction, it must await the consensus result from consensus nodes before executing the transaction. This results in long-term idle waste of computational resources of full nodes, exacerbating the end-to-end transaction latency and degrading the QoS. SMP-based protocols have achieved significant throughput improvements by decoupling transaction distribution from consensus. However, higher throughput, which

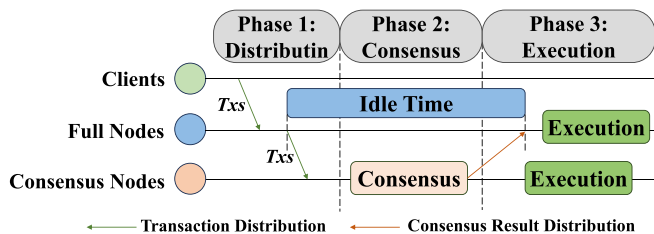


Fig. 3. The “DiCE” workflow.

means more transactions in a single consensus round, also inevitably requires a longer transaction execution time under the DiCE workflow. These SMP-based protocols have failed to consider the consequential impact of increased throughput on transaction execution.

Further analysis reveals that the block substructures produced by consensus nodes in SMP-based protocols [11], [12], [13], [14], [15] are essentially ordered results of partial transactions from the transaction pool of each consensus node. In these protocols, generating a new block proposal is essentially a straightforward encapsulation of the block substructures produced by each consensus node, where the encapsulation order matches the generation order. This process does not disrupt the order of transactions within each block substructure. Additionally, the transaction partitioning mechanism in *Nexus* ensures that each client’s transactions are processed by only one consensus node, thereby eliminating duplicate transactions across block substructures generated by different consensus nodes. Based on these properties, the **Pre-Execution** mechanism is proposed in *Nexus*. This mechanism decouples transaction execution from consensus and fully utilizes the otherwise idle computational resources of full nodes while waiting for consensus results, thereby accelerating the transaction execution phase and significantly reducing the end-to-end latency.

Specifically, when a consensus node generates new block substructures and forwards them to other consensus nodes, it can simultaneously forward these substructures to its affiliated full nodes. Similarly, when a consensus node receives new block substructures from other consensus nodes, it can also forward these in parallel to the full nodes affiliated with itself. Upon receiving block substructures, the full nodes can pre-execute the transactions following the specified order within each block substructure. This process generates a set of state incremental modifications that await final confirmation, rather than directly modifying the state tree to generate a new state. The details of the set of state incremental modifications are provided in Section III-D. Once the full nodes receive the final consensus result, they follow the order of block substructures within the block and sequentially extract those block substructures with the same hash values from their pre-executed block substructures.

If the block substructure with the same hash value is found, the set of state incremental modifications of the corresponding block substructure can be finally confirmed and updated to the state tree. If not found, the node can request the missing block substructure from other consensus nodes based on its hash value and re-execute it to obtain the correct set of state modifications.

Therefore, even if a malicious consensus node deliberately does not forward the block substructure or forwards an incorrect block substructure, it cannot compromise the safety and liveness of the system. Furthermore, during the process of merging the pre-execution results of block substructures, if state conflicts are detected, it is only necessary to re-execute the transactions related to the conflicting states within the block substructure, rather than re-executing the entire block substructure.

C. Feedback on Execution

The SMP-based protocol allows for a large number of transactions to reach consensus at once. However, for permissioned blockchains with limited resources, this can significantly prolong the transaction execution time, thereby severely impacting both throughput and latency. This performance limitation occurs because consensus nodes must execute all transactions before proceeding with consensus and generating new block proposals. To reduce the computational pressure on consensus nodes during the transaction execution phase, a mechanism named **Feedback on Execution** is proposed in *Nexus*. Its core idea is to allow full nodes to sign and send the pre-execution results of block substructures to their affiliated consensus node. Subsequently, the consensus node distributes these results to other consensus nodes.

According to the system model in *Nexus*, there are at most f malicious nodes among full nodes. To prevent erroneous pre-execution results submitted by malicious full nodes, each consensus node randomly connects to $f + 1$ full nodes and concurrently collects their pre-execution results of block substructures. If the pre-execution results from these $f + 1$ full nodes for the same block substructure are consistent, the correctness of the pre-execution result is established. The consensus node then sends this pre-execution result along with the signatures from these full nodes to other consensus nodes. Upon receiving this signed pre-execution result, the other consensus nodes forward it to full nodes affiliated with them. If the pre-execution results from these $f + 1$ full nodes for the same block substructure are inconsistent, it indicates that there are malicious nodes among the current $f + 1$ full nodes. In this case, the consensus node randomly selects an additional f full nodes and requests their pre-execution results of the block substructure. As a result, the consensus node will collect $2f + 1$ pre-execution results of the same block substructure. Since there are at most f malicious full nodes in the system, the majority result among these $2f + 1$ pre-execution results is guaranteed correct. Therefore, the consensus node can safely adopt this majority result. For subsequent block substructures, it selects $f + 1$ full nodes from those that returned the correct result to continue collecting pre-execution results. Regarding the impact of malicious full nodes, this reselection method guarantees that the consensus node needs to perform the aforementioned process at most once, which constitutes a single retry, to obtain $f + 1$ full nodes capable of providing consistent pre-execution results. Consequently, the consensus node avoids indefinite waiting periods. In this process, since each full node independently computes pre-execution results and the consensus node only interacts in a single round with the additional f full

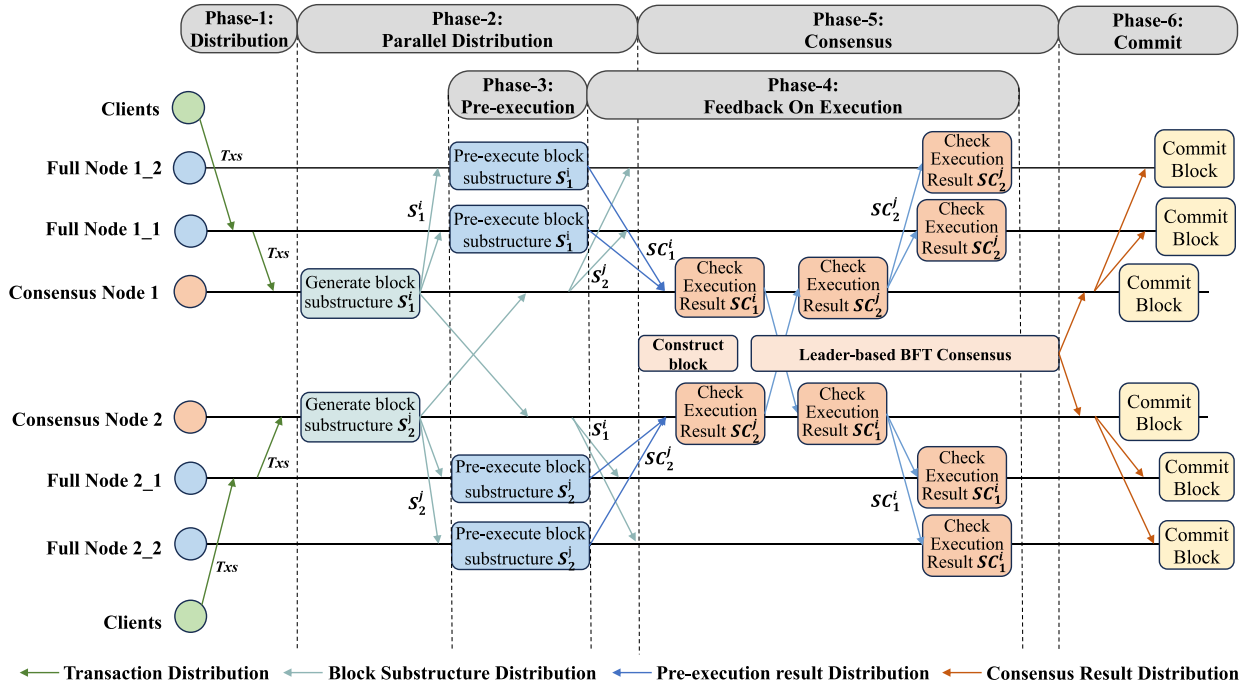


Fig. 4. Workflow of Nexus.

nodes, the extra time cost is approximately one Round Trip Time (RTT) plus the necessary verification time. This ensures that both time and bandwidth overheads remain acceptable, thereby maintaining the efficiency of *Nexus*.

When consensus nodes reach consensus on a new block proposal, they, just like the full nodes described in Section III-B, need only to reorganize the corresponding block substructures and then sequentially merge their pre-execution results. If during the reorganization process, a consensus node finds that the pre-execution result of a certain block substructure is missing, it only needs to re-execute the corresponding block substructure. If a state conflict is detected, it only needs to re-execute the transactions related to the conflicting state in the block substructure.

Through this mechanism, full nodes affiliated with different consensus nodes can share the pre-execution results of block substructures. Therefore, each full node needs only to pre-execute the block substructures generated by its affiliated consensus node. This typically accounts for only $\frac{1}{N_c}$ of the total transactions. This significantly lowers the computational overhead for both full nodes and consensus nodes. Such an approach is especially advantageous for permissioned blockchains with limited resources.

D. Workflow of Nexus

Fig. 4 shows the workflow of *Nexus* with six phases.

Phase-1 Distribution: A client continuously generates transactions in the form of $Tx_n = [pk, (T_p, T_s, N_d, \Omega)_\sigma]$, where pk is the public key and the address of the client, T_p is the transaction payload, T_s is the sequence number of the transaction, and N_d is the consensus node responsible for processing transactions of the client as determined by the transaction partitioning

mechanism. Ω is a collection of signatures used when a client wishes to change the consensus node responsible for handling its transactions. σ is the signature generated by using the client's private key over the transaction content. After the transaction is generated, the client forwards it to a full node affiliated with N_d , and the full node subsequently forwards the transaction to N_d .

Phase-2 Parallel Distribution: This phase is divided into two parts. The first part is generating block substructures. Since *Nexus* is an optimization for SMP-based protocols with emphasis on transaction execution, the block substructure can be simplified to a transaction collection. Specifically, $S_i^k = \langle Tx_1, Tx_2, \dots, Tx_n \rangle$ denotes the k -th block substructure generated by consensus node i . The second part is the distribution of block substructures. As described in the previous subsection, consensus nodes concurrently distribute these substructures to other consensus nodes and full nodes affiliated with them. The data distribution algorithm between consensus nodes and full nodes uses erasure coding (EC), which can minimize the bandwidth consumption of consensus nodes when distributing data to full nodes.

Phase-3 Pre-Execution: When a full node receives a block substructure S_i^k generated by its affiliated consensus node i , it executes the transactions within S_i^k sequentially. This pre-execution generates a set of state incremental modifications to the variables of the world state, denoted as $SC_i^k = [(\alpha_1, \beta_1, \gamma_1), \dots, (\alpha_j, \beta_j, \gamma_j)]$. Here, α refers to the variable implicated in transactions within S_i^k , while β represents the alteration of the corresponding variable after executing all transactions within S_i^k . γ represents the initial on-chain state required by transactions with dynamic execution logic in S_i^k that modify α . If all transactions involving modifications to α follow static execution logic, this value is set to null. The

definitions of dynamic and static execution logic will be introduced in Section III-G. Importantly, SC_i^k is not the new state after executing the transactions in S_i^k ; rather, it represents incremental changes to these variables. Subsequently, the full node signs this modification set SC_i^k and sends it back to its affiliated consensus node i .

Phase-4 Feedback on Execution: After a consensus node collects the pre-execution results from $f + 1$ full nodes affiliated with it for a block substructure it generated, it initiates the ‘‘Check Execution Result’’ process. This process verifies the set of state modifications SC_i^k , received from these $f + 1$ full nodes. If all $f + 1$ instances of SC_i^k are identical, it indicates the correctness of SC_i^k . Subsequently, the consensus node broadcasts SC_i^k together with $f + 1$ signatures of it to the other consensus nodes. The other consensus nodes relay SC_i^k to the full nodes affiliated with them. If they differ, the consensus node will reselect full nodes according to the algorithm described in Section III-C.

Phase-5 Consensus: The leader proposes a new block proposal $Proposal_{new}$, which consists of the block header and hashes of block substructures. The leader runs a leader-based BFT consensus algorithm, such as PBFT or HotStuff, to reach consensus with the other consensus nodes on $Proposal_{new}$. Once consensus is achieved, the consensus nodes distribute $Proposal_{new}$ to full nodes and proceed to the Commit phase.

Phase-6 Commit: Nodes, including both full nodes and consensus nodes, extract each S_i^k from their pre-executed block substructures according to the order of block substructures specified in $Proposal_{new}$. If S_i^k is found, the node merges the corresponding pre-execution results SC_i^k into the state tree. Otherwise, the node must re-execute all transactions within that block substructure. In this case, the parallel workflow reverts to a sequential workflow. This situation may arise if some full nodes or consensus nodes behave maliciously. This issue can be addressed by blocklisting malicious consensus nodes or by reselecting the full nodes. Furthermore, during the merging of pre-execution results, if it is found that the initial state (recorded in γ_i) on which the dynamic execution logic transactions that modify α_i depend is inconsistent with the current state, this indicates a conflict has been detected. Consequently, the relevant transactions in the block substructure will be re-executed based on the current state.

E. Performance Analysis of Nexus

This subsection provides a TPS analysis of *Nexus*. Let T denote the number of transactions that can reach consensus per unit time, excluding the transaction execution phase. Furthermore, the average execution time for a transaction is assumed to be t_e seconds, and a proportion m of the transactions processed during this unit require complete execution by the consensus nodes. Therefore, the theoretical maximum TPS considering the transaction execution is presented in (1).

$$TPS = \frac{T}{1 + T \cdot t_e \cdot m} \quad (1)$$

As shown in (1), a critical factor affecting the TPS of *Nexus* is m , which represents the proportion of transactions that require

execution among all transactions. By default, $m = 1$ indicates that each transaction requires execution. Since T in these SMP-based protocols typically reaches tens of thousands, and the average execution time t_e for transactions is generally in the range from hundreds of microseconds to tens of milliseconds [1], [32], [33], the value of $T \cdot t_e$ is significantly greater than 1. Under this condition, the TPS can be approximated by (2).

$$TPS = \frac{T}{1 + T \cdot t_e \cdot m} \approx \frac{T}{T \cdot t_e} = \frac{1}{t_e} \quad (2)$$

According to (2), when $m = 1$, the upper bound of TPS is approximately the reciprocal of the average execution time of a single transaction, t_e . Based on the value of t_e , the actual TPS is just under 2000 tx/s, which is 8 to 15 times lower than the TPS reported in prior works [11], [12], [13], [14], [15]. This discrepancy arises because those studies did not take transaction execution into account during TPS testing. *Nexus* decouples transaction execution from consensus, enabling full nodes to pre-execute transactions via the parallel distribution of block substructures and to feed the results back to the consensus nodes. *Nexus* can minimize the number of transactions that consensus nodes need to execute, thereby effectively reducing the value of m . The primary performance bottleneck in *Nexus* is the number of transactions that a full node can process within a unit of time. However, full nodes in *Nexus* are only responsible for executing the block substructures generated by their affiliated consensus nodes. Therefore, as the number of consensus nodes increases, the transaction load on each consensus node decreases. This allows full nodes with sufficient time to pre-execute all transactions in the block substructures generated by their affiliated consensus nodes. For consensus nodes, this means $m = 0$, and the TPS of *Nexus* under such circumstances is as described in (3). These conclusions will be experimentally validated in Section IV.

$$TPS = \frac{T}{1 + T \cdot t_e \cdot 0} = T \quad (3)$$

According to (3), in an ideal scenario, *Nexus* can eliminate the negative impact of transaction execution, thereby increasing the TPS to levels reported in papers on SMP-based protocols. However, the experiments described in Section IV are unlikely to achieve such optimal performance. This limitation arises because the communication between consensus nodes and full nodes consumes bandwidth, and the additional distribution of pre-execution results among consensus nodes also consumes bandwidth despite each result being only 1-3 KB, as each block substructure has a maximum of 50 transactions. These factors prevent the full utilization of available bandwidth for generating block substructures.

F. Proof of Nexus

To prove the safety of *Nexus*, two symbols are defined: S_{li}^k denotes the k -th block substructure received by the consensus node i from the consensus node l . SC_{li}^k denotes the pre-execution result of S_{li}^k received by the consensus node i from the consensus node l .

Lemma 1. Block Substructure Consistency: For nodes i and j , if S_{li}^k and S_{lj}^k are valid, then $S_{li}^k = S_{lj}^k$.

Proof. In SMP-based protocols [11], [12], [13], [14], [15], this property has been proven. Since *Nexus* does not modify the generation and distribution mechanisms of block substructures between consensus nodes, this can be directly adopted as a lemma.

Theorem 1. Pre-Execution Consistency: For nodes i and j , if SC_{li}^k and SC_{lj}^k are correct, then $SC_{li}^k = SC_{lj}^k$.

Proof. Consensus nodes cannot forge the signatures of full nodes and can only either forward or not forward the pre-execution results received from full nodes. Therefore, for the correct pre-execution results of the same block substructure, they satisfy $SC_{li}^k = SC_{lj}^k$.

Theorem 2. Order Consistency: If the states of nodes i and j are $State_i$ and $State_j$, and satisfy $State_i = State_j$, then consider the following: nodes i and j merge the pre-execution results of S_l^k and S_q^h , which are SC_{li}^k, SC_{qi}^h for node i and SC_{lj}^k, SC_{qj}^h for node j . Node i merges SC_{li}^k first and then SC_{qi}^h , resulting in the state $State_{i_1}$. Node j merges in the same order resulting in the state $State_{j_1}$. It satisfies $State_{i_1} = State_{j_1}$.

Proof. According to *Theorem 1*, $SC_{li}^k = SC_{lj}^k = SC_l^k$ and $SC_{qi}^h = SC_{qj}^h = SC_q^h$. The pre-execution results SC_{li}^k and SC_{qi}^h are sets of state incremental modifications obtained after pre-executing the block substructures S_l^k and S_q^h rather than a new state. Consequently, for node i and node j , the states $State_{i_1}$ and $State_{j_1}$, obtained by merging SC_l^k and SC_q^h in the same order, must satisfy $State_{i_1} = State_{j_1}$.

Theorem 3. State Consistency: If the states of nodes i and j are $State_i$ and $State_j$ respectively, and satisfy $State_i = State_j$, then after executing all transactions in the identical block $Block_{new}$ by merging the pre-execution results in the same order, the new states $State_{i_{new}}$ and $State_{j_{new}}$ of nodes i and j will satisfy $State_{i_{new}} = State_{j_{new}}$.

Proof. According to *Lemma 1* and *Theorem 1*, $S_{li}^k = S_{lj}^k, SC_{li}^k = SC_{lj}^k, \forall S_l^k \in Block_{new}$. Given that the initial states satisfy $State_i = State_j$, and combining the order consistency proved in *Theorem 2*, it can be concluded that after nodes i and j execute the same block through merging the identical pre-execution results in the same order, the new states $State_{i_{new}}$ and $State_{j_{new}}$ must satisfy $State_{i_{new}} = State_{j_{new}}$.

Theorem 4. Error Result Non-commitment: If a malicious full node p generates an invalid pre-execution result SC_j^{tk} for the block substructure S_j^k , this result will not be submitted by any honest consensus nodes.

Proof. *Nexus* requires consensus nodes to connect to $f + 1$ full nodes to collect pre-execution results. Since there are at most f malicious nodes in the system, at least one honest node will submit the correct pre-execution result. According to the design of *Nexus*, the pre-execution results from $f + 1$ full nodes will be accepted by consensus nodes only if they are identical. Therefore, an invalid pre-execution result SC_j^{tk} will not be submitted. Additionally, if an inconsistency is detected, the consensus nodes will re-execute the transactions in the corresponding block substructure locally.

Nexus serves as a performance optimization and does not harm the liveness of the system. Consider the following failure scenarios: (1) malicious full nodes provide invalid pre-execution

results, (2) full nodes do not produce any results, or (3) consensus nodes do not use or distribute received results. In any of these cases, the only consequence is that the transactions in the affected block substructure must be re-executed locally by consensus nodes. The consensus will continue to advance and not harm the liveness of the system.

G. Transactions in Nexus

This subsection analyzes the transaction processing mechanism of *Nexus* and discusses it for different transaction types. *Nexus* employs a transaction partitioning mechanism that assigns transactions to different consensus nodes based on the address of the client. Each consensus node then generates block substructures and distributes them to its affiliated full nodes for pre-execution. The consensus nodes collect these pre-execution results and share them with other consensus nodes.

As described in Phase-3 of Section III-D, during the pre-execution of block substructures, the results generated by full nodes constitute a set of state incremental modifications, rather than a new state. Since the pre-execution results of different block substructures are independent, transactions whose state incremental modifications in pre-execution results are guaranteed to be consistent with those under a global order must have static execution logic. That is, the computation of their state modifications depends only on transaction parameters and not on the current on-chain state. Therefore, regardless of global ordering, the state incremental modifications produced during pre-execution are always deterministic. The transactions with static execution logic include transfer transactions, asset movements, and independent operations on smart contract state variables without cross-dependencies (such as the insertion, deletion, or modification of mappings, sets, and numerical variables). For these types of transactions, the pre-execution results in *Nexus* are static, allowing the full nodes to efficiently generate the correct set of state incremental modifications. Our analysis of over 124 million real Ethereum transactions [34] indicates that such transactions constitute approximately 83% of the total transactions.

For transactions with dynamic execution logic, their computation depends on the current on-chain state, such as conditional branches in smart contracts. Due to *Nexus*'s transaction partitioning mechanism, full nodes pre-execute block substructures based on their local state. As a result, they may not be aware of state changes caused by transactions assigned to other consensus nodes. Consequently, the state incremental modifications generated via pre-execution may conflict with those generated under the global order. When such conflicts are detected during the commit phase, *Nexus* will serially re-execute the relevant transactions in corresponding block substructures to ensure the consistency and correctness of the state. It should be noted that not all transactions with dynamic execution logic result in conflicts. The most typical conflicting scenario arises when, within the same block, transactions assigned to different consensus nodes according to the transaction partitioning mechanism invoke the same smart contract function with dynamic execution logic. Our analysis of over 124 million real Ethereum transactions shows

that, with 16 consensus nodes, such transactions constitute only 4.7% of the total transactions.

Additionally, it should be emphasized that the merging order of pre-execution results depends on the order of the corresponding block substructures within the block, as merging the pre-execution results in different orders may generate different states. Consider a scenario where block substructure S_j^k contains transaction $Tx_1: A \rightarrow B$ 10ETH, and block substructure S_i^h contains transaction $Tx_2: B \rightarrow C$ 10ETH. If the order in block is S_j^k, S_i^h , both transactions can be successfully committed. However, if the order in block is S_i^h, S_j^k , the Tx_2 in S_i^h would fail to commit. The occurrence of such conflicts is not due to the design of *Nexus*. Since *Nexus* focuses on the transaction execution layer, its responsibility is limited to ensuring that transactions are executed strictly in the order specified within the block. The responsibility for deciding transaction inclusion and ordering within blocks resides entirely with the transaction packaging algorithm used in the consensus layer. This algorithm has been implemented in these SMP-based protocols [11], [12], [13], [14], [15], and its design falls beyond the scope of this work.

IV. EVALUATION

Predis [12] is implemented on HotStuff [27], and *Nexus* is further implemented on *Predis*. The performance of *Nexus* is evaluated using Alibaba Cloud ECS. Each ECS instance is ecs.c7.2xlarge, equipped with 8 cores and 16 GB of memory. NetEm [35] is used to simulate a WAN environment with 30 ms inter-node RTT and 100 Mb/s bandwidth for each node in a LAN. Results are averaged over five independent runs. Each block substructure contains 50 transactions of 512 bytes. Each consensus node connects to $f + 1$ full nodes, and all clients generate transfer transactions at identical rates. Throughout the evaluation, latency refers to the time from when the client submits a transaction to when the client receives a response.

Baseline: *Nexus* is compared with *Predis* [12], as *Predis* shows the best performance among existing SMP-based protocols. In *Predis*, the block substructures generated by the same consensus node are organized in a chain-like structure, which reduces the complexity of bandwidth consumption in block proposal distribution from linear to constant. Since the performance tests in the *Predis* paper do not include transaction execution phase, this phase is added to *Predis*, referred to as *Predis with Transaction Execution (Predis with TE)*. *Nexus* is then compared with *Predis with TE* to demonstrate its effectiveness. Additionally, *Nexus* is compared with the state-of-the-art blockchain parallel processing framework, *Bidl* [36], which is also implemented on *Predis* to ensure a fair comparison and to highlight the advancements of *Nexus*. *Bidl* introduces a special node, the “Sequencer”, to serve as the gateway for all transactions. The “Sequencer” assigns a unique sequence number to each transaction and distributes the sequenced transactions to all consensus nodes and full nodes. Subsequently, other nodes can execute transactions according to these sequence numbers.

This section presents the results of three experiments to address the following research questions:

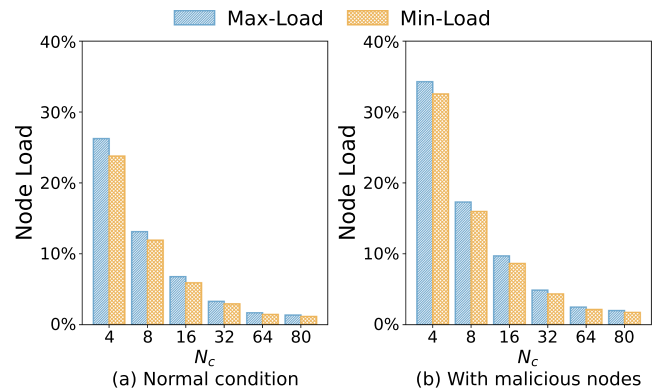


Fig. 5. Max-Min node load under different scales.

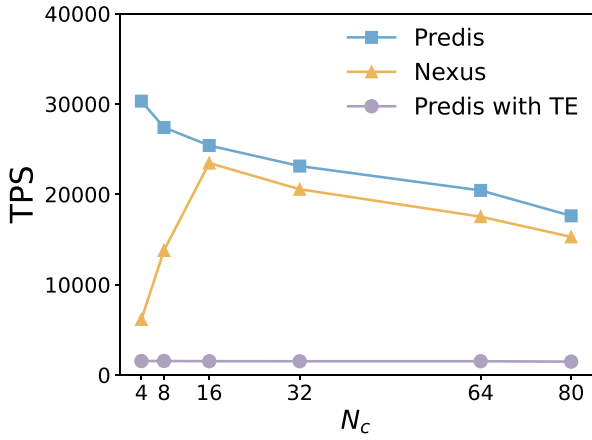
- *RQ1*: Is the transaction partitioning mechanism in *Nexus* effective, and does it achieve load balancing?
- *RQ2*: How does *Nexus* perform compared to baseline systems in terms of throughput, latency, and quality of service (QoS)?
- *RQ3*: How does *Nexus* perform under attacks from malicious nodes?

A. Transaction Partitioning Mechanism

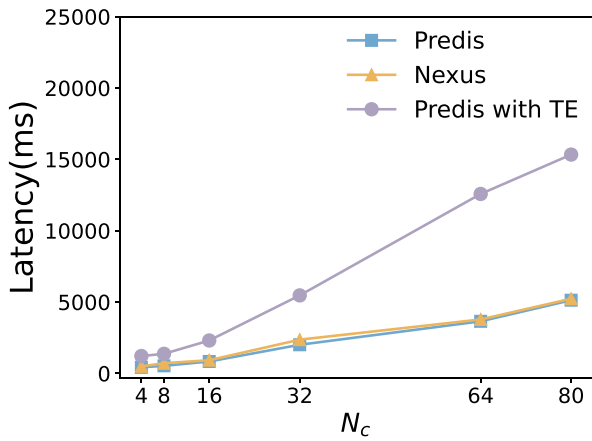
Nexus proposes a transaction partitioning mechanism based on consistent hashing to prevent the “transaction duplication” attack and achieve load balancing between clients and consensus nodes. Considering computational efficiency, *Nexus* employs the *FNV1_32_HASH* [37] algorithm to map consensus nodes and clients onto the hash ring. Based on practical experience, *Nexus* adds 1,000 virtual nodes for each consensus node. To verify the effectiveness of this mechanism, the transaction load distribution is evaluated across consensus nodes at different scales, with the number of consensus nodes set to 4, 8, 16, 32, 64, and 80.

Fig. 5(a) shows that the transaction load distribution among consensus nodes is relatively balanced at different consensus node scales, with no consensus node being overburdened or underutilized. In the scenario with 4 consensus nodes, the disparity in load among consensus nodes is only about 2.5%. Moreover, the load discrepancy among consensus nodes progressively diminishes as the number of consensus nodes increases. Our experiments show that each transaction is allocated to the clockwise-closest consensus node on the hash ring, and no transaction is added to multiple transaction pools simultaneously. These empirical results demonstrate that *Nexus*’s transaction partitioning mechanism effectively prevents “transaction duplication” attacks while ensuring load balancing between clients and consensus nodes.

Furthermore, Fig. 5(a) illustrates that as the number of consensus nodes increases, the load percentage of each consensus node gradually decreases. This observation corroborates the conclusion presented in Section III: as the number of consensus nodes increases, the number of transactions that full nodes need to pre-execute decreases since they only need to pre-execute the block substructures produced by consensus nodes they are affiliated with.



(a) TPS of Nexus vs Predis.



(b) Latency of Nexus vs Predis.

Fig. 6. Throughput and Latency of Nexus.

To further verify whether this mechanism can maintain load balancing in the presence of malicious consensus nodes, the transaction load distribution is evaluated across consensus nodes at different scales, with malicious nodes constituting 1/3 of the total consensus nodes at each scale. Fig. 5(b) illustrates our experimental results. The results indicate that in the presence of malicious consensus nodes, the load of each consensus node slightly increases compared to normal conditions. This is because the remaining consensus nodes must share the load initially carried by the malicious nodes. However, relative load balancing among consensus nodes is still achieved, demonstrating the robust fault tolerance and scalability of the transaction partitioning mechanism in Nexus.

B. Throughput and Latency of Nexus

First, the performance of Nexus and Predis is evaluated at different scales of consensus nodes. As illustrated in Fig. 6(a) and (b), Nexus demonstrates significant performance advantages over Predis with TE: throughput increases by 4× to 15× (varying with system scale), while latency decreases by 50% to 70%. These substantial improvements clearly demonstrate the effectiveness of Nexus.

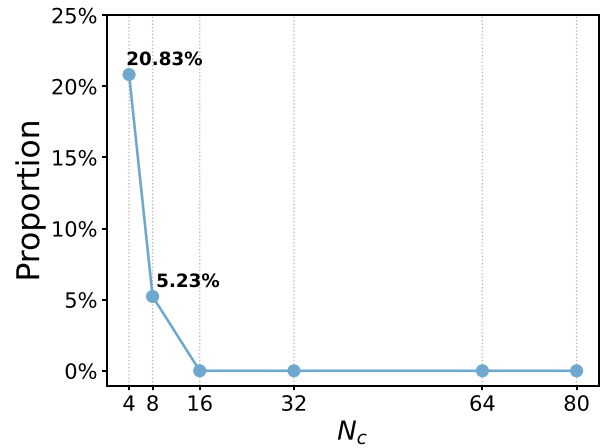


Fig. 7. Transaction execution proportion at different scales.

Additionally, Fig. 6(a) and (b) also show that when the number of consensus nodes is fewer than 16, the TPS of Nexus significantly lags behind Predis but still remains higher than Predis with TE. Combining this with Fig. 7, which illustrates the percentage of transactions that need to be executed by consensus nodes at different scales, it can be observed that each node must handle a significant number of transactions in scenarios with a small scale of consensus nodes. Consequently, the number of transactions in block substructures generated by consensus nodes in a consensus round exceeds the execution capacity of the full nodes as the transaction generation rate increases. Therefore, consensus nodes must execute transactions not pre-executed by full nodes, thereby extending the duration of a consensus round and consequently decreasing throughput.

When the number of consensus nodes exceeds 16, the TPS of Nexus is very close to that of Predis. Combining this with Fig. 7, it is evident that consensus nodes no longer need to execute any transactions in these scenarios. This demonstrates that Nexus has addressed the limitation in SMP-based protocols caused by the lengthy transaction execution time due to a large number of transactions confirmed in a single consensus round. However, the TPS of Nexus remains slightly lower than that of Predis in these scenarios. This is due to the additional bandwidth consumed for distributing pre-execution results among consensus nodes and between consensus nodes and full nodes, as previously analyzed in Section III-E. Nevertheless, the TPS difference between Nexus and Predis is minimal, which confirms that the additional bandwidth overhead is small. The above experimental results conclusively demonstrate that Nexus is particularly well-suited for large-scale consensus node environments, exhibiting exceptional scalability.

Nexus is also compared with Bidl, a state-of-the-art blockchain system with transaction pre-execution capabilities. As illustrated in Fig. 8(a) and (b), Nexus exhibits consistently lower transaction latency than Bidl across all tested node scales. In terms of TPS, Nexus significantly outperforms Bidl. Moreover, as the number of consensus nodes increases, the advantage of Nexus becomes even more pronounced. This growing performance gap can be attributed to an architectural limitation

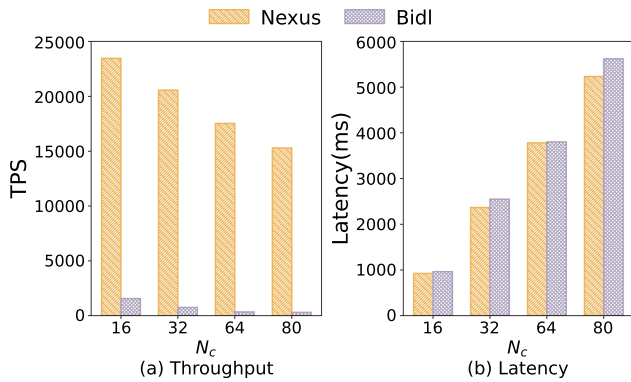


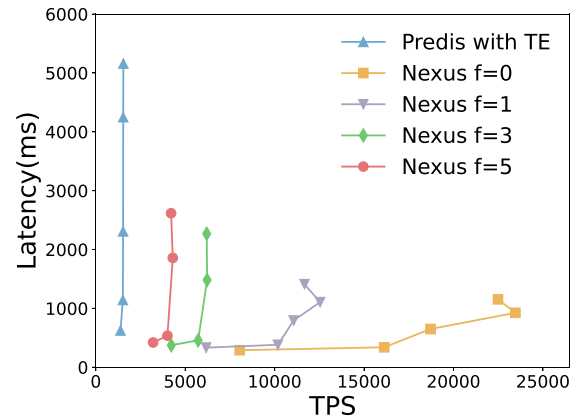
Fig. 8. Throughput and Latency of *Nexus* vs. *Bidl*.

of *Bidl*: it relies on a single sequencer as the centralized entry point for all transactions, which creates a bottleneck because the sequencer must forward each transaction to all consensus nodes and all full nodes. As the number of consensus nodes increases, the limited bandwidth of the sequencer leads to a reduction in the number of effective transactions distributed to each node, thereby decreasing the overall TPS. In contrast, *Nexus* allows all consensus nodes to serve as entry points for transactions, and they forward the block substructures they generate to each other. Furthermore, due to the Feedback on Execution mechanism in *Nexus*, each node only needs to execute $1/N_c$ of all transactions, whereas nodes in *Bidl* are still required to execute all transactions. The comparison with *Bidl* demonstrates that *Nexus* achieves higher TPS and lower latency, and is particularly well suited to permissioned blockchains with limited resources.

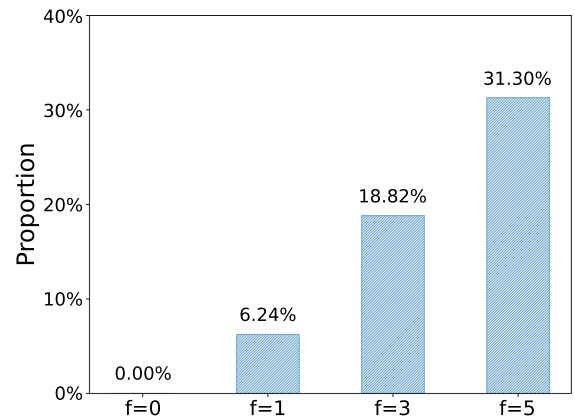
C. Nexus Under Malicious Attack

There are two typical scenarios in which malicious nodes cause a throughput decline in *Nexus*: (1) malicious full nodes either not executing or submitting incorrect pre-execution results, and (2) malicious consensus nodes not forwarding the pre-execution results from full nodes to other consensus nodes. Although these malicious behaviors differ in specifics and are carried out by different types of nodes, they result in the same issue: the block substructures handled by malicious nodes lack valid pre-execution results, forcing consensus nodes to re-execute transactions within those block substructures. With a maximum of f malicious nodes in the system, the consensus nodes may have to re-execute the block substructures generated by up to $\frac{f}{N_c}$ proportion of consensus nodes. To eliminate the impact on performance due to the transaction execution capacities of full nodes, the throughput and latency of *Nexus* are tested with 16 consensus nodes, considering scenarios with 1, 3, and 5 malicious nodes.

Fig. 9(b) illustrates the proportion of transactions that consensus nodes need to re-execute relative to the total number of transactions under different numbers of malicious nodes. As the number of malicious nodes increases, the proportion of transactions requiring re-execution by consensus nodes also increases from 0, approximately at a rate of $\frac{1}{16}$. Fig. 9(a) compares the throughput and latency between *Nexus* under different numbers of malicious nodes and the *Predis with TE*. It is evident that as the



(a) TPS and Latency.



(b) Transaction execution proportion.

Fig. 9. Throughput and Latency Under Malicious Attack.

number of malicious nodes increases, the proportion of transactions that consensus nodes need to execute increases, leading to a decrease in *Nexus*'s throughput and an increase in transaction latency. The observed throughput degradation pattern quantitatively validates our theoretical performance model for *Nexus* as formulated in (1), thereby confirming the accuracy of our analytical framework. Fig. 9(a) shows that even with malicious nodes present, *Nexus* still maintains safety and liveness while achieving higher throughput and lower latency compared to the *Predis with TE*.

V. RELATED WORK

A. Blockchain Transactions Processing Workflow

Hyperledger Fabric [1] is a permissioned blockchain framework that follows the "Execution-Distribution-Consensus" workflow with a strictly linear transaction lifecycle: transactions must complete the endorsement phase before consensus, meaning the block generation is solely based on transactions that have completed execution, and the system does not include transactions with incomplete endorsements in blocks. In contrast, *Nexus* is designed for permissioned blockchains that follow the "Distribution-Consensus-Execution" workflow, aiming to address performance bottleneck issues in SMP-based protocols [11], [12], [13], [14], [15], [16]. *Nexus* implements

parallelization optimizations that decouple transaction execution from consensus, thereby enabling transaction execution to proceed in parallel with consensus. Consequently, *Nexus* allows blocks to contain transactions that have not yet completed pre-execution, significantly improving the transaction processing efficiency.

B. Parallelization Execution

The concept of parallelization has already been applied in blockchains. Vikram [38] analyzes the effectiveness of parallel execution of transactions within each block of Ethereum. Eve [39] allows nodes to execute a batch of requests in parallel optimistically. LazyLedger [40] fully delegates transaction execution to application clients. Consensus nodes no longer maintain the ledger state or provide state finality, and only offer data availability. This results in the absence of a unified global state. This forces new users to synchronize from the genesis block, incurring high costs. Furthermore, since each application in LazyLedger operates in an isolated state space, cross-application interactions require complex dependency management and must be pre-hardcoded, constraining interoperability. DispersedLedger [41] focuses on mitigating performance bottlenecks caused by bandwidth heterogeneity among nodes in WAN. It allows consensus nodes to agree only on block digests and then independently download block data and execute transactions. This approach does not address the bottleneck of transaction execution, as each node must still independently bear the computational cost of executing all transactions. Essentially, this remains a serial execution model.

The work most closely related to *Nexus* is *Bidl* [36], which is designed for the Hyperledger Fabric [1]. It introduces a “Sequencer” node to sequence transactions and distribute the sequenced transactions to all consensus nodes and full nodes. Once nodes receive these sequenced transactions, they can execute them in order, achieving parallelization between transaction execution and consensus. *Nexus* has several advantages over *Bidl*: (1)*Nexus* eliminates the “Sequencer” node, avoiding single points of failure. (2)*Nexus* allows each consensus node to serve as a transaction entry point, unlike *Bidl*’s reliance on the “Sequencer”, which requires extensive bandwidth and computational resources to prevent the performance bottleneck, as shown in Section IV. (3)*Nexus* allows for sharing pre-execution results among nodes. Each node processes only $\frac{1}{N_c}$ of total transactions, in contrast to *Bidl*, where each node must execute all transactions. (4)*Nexus* pre-executes at block substructure granularity and produces state modification sets, requiring only conflicting substructures to be re-executed, while *Bidl* pre-executes at single transaction granularity and produces new state, requiring re-executing all transactions following any conflict point. These advantages make *Nexus* more suitable for resource-limited permissioned blockchains.

Other research explores transaction concurrency from the perspective of individual nodes to reduce the proportion of time spent on the transaction execution phase within a single block’s lifecycle. This research can be divided into parallel execution models based on static analysis [42], [43] and dynamic analysis [44], [45], [46], [47]. These approaches are orthogonal to

Nexus, as they investigate how transactions within a single block can be concurrently executed using multiple threads through a transaction dependency graph. Therefore, they can be used in conjunction with *Nexus* to enhance the efficiency of full nodes in executing transactions within block substructures.

C. Sharding and Multi-Chain

Numerous research efforts have taken a system perspective to enhance blockchain performance through sharding or multi-chain. Sharding [48], [49], [50], [51], [52] works by dividing the blockchain nodes into multiple shards, with each shard processing a subset of transactions. However, a drawback of sharding is that dispersing consensus nodes makes them more vulnerable to attacks. Multi-chain [53], [54] involves a main chain and several subsidiary chains. Each chain has its independent consensus group. Multi-chain increases throughput by distributing transactions across different subsidiary chains. *Nexus*, as a performance optimization framework designed for SMP-based protocols, is orthogonal to sharding and multi-chain. *Nexus* can be applied to these subsidiary chains or shards to further enhance their performance.

VI. CONCLUSION

This paper has addressed the critical performance bottlenecks in permissioned blockchains that adopts SMP-based protocols by proposing *Nexus*, which decouples transaction execution from consensus. *Nexus* efficiently utilizes the idle computational resources of full nodes and shortens the transaction execution time for all nodes, enhancing throughput by 4x to 15x and reducing latency by 50% to 70%. Furthermore, *Nexus* implements an efficient transaction partitioning mechanism, which addresses the “transaction duplication” attack and achieves load balancing. While *Nexus* significantly improves QoS, future work should explore adaptive partitioning mechanisms and cross-chain transaction support to further extend its applicability, as well as methods to mitigate potential conflicts arising from transactions with dynamic execution logic.

VI. ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their comments.

REFERENCES

- [1] E. Androulaki et al., “Hyperledger fabric: A distributed operating system for permissioned blockchains,” in *Proc. 13th EuroSys Conf.*, 2018, pp. 1–15.
- [2] M. Nofer, P. Gomber, O. Hinz, and D. Schiereck, “Blockchain,” *Bus. Inf. Syst. Eng.*, vol. 59, pp. 183–187, 2017.
- [3] M. Castro et al., “Practical byzantine fault tolerance,” *OsDI*, vol. 99, no. 1999, pp. 173–186, 1999.
- [4] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, “HotStuff: BFT consensus with linearity and responsiveness,” in *Proc. ACM Symp. Princ. Distrib. Comput.*, 2019, pp. 347–356.
- [5] G. G. Gueta et al., “SBFT: A scalable and decentralized trust infrastructure,” in *Proc. 49th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2019, pp. 568–580.
- [6] B. Y. Chan and E. Shi, “Streamlet: Textbook streamlined blockchains,” in *Proc. 2nd ACM Conf. Adv. Financial Technol.*, 2020, pp. 1–11.
- [7] E. Buchman, J. Kwon, and Z. Milosevic, “The latest gossip on BFT consensus,” 2018, *arXiv:1807.04938*.

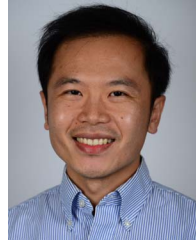
- [8] C.-D. Liu-Zhang, C. Matt, and S. E. Thomsen, "Asymptotically optimal message dissemination with applications to blockchains," in *Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2024, pp. 64–95.
- [9] I. Kaklamani, L. Yang, and M. Alizadeh, "Poster: Coded broadcast for scalable leader-based BFT consensus," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2022, pp. 3375–3377.
- [10] J.-S. Kim, J.-M. Shin, S.-H. Choi, and Y.-H. Choi, "A study on prevention and automatic recovery of blockchain networks against persistent censorship attacks," *IEEE Access*, vol. 10, pp. 110770–110784, 2022.
- [11] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and tusk: A DAG-based mempool and efficient BFT consensus," in *Proc. 17th Eur. Conf. Comput. Syst.*, 2022, pp. 34–50.
- [12] Z. Hu et al., "A data flow framework with high throughput and low latency for permissioned blockchains," in *Proc. IEEE 43rd Int. Conf. Distrib. Comput. Syst.*, 2023, pp. 1–12.
- [13] F. Gai, J. Niu, I. Beschastnikh, C. Feng, and S. Wang, "Scaling blockchain consensus via a robust shared mempool," in *Proc. IEEE 39th Int. Conf. Data Eng.*, 2023, pp. 530–543.
- [14] K. Hu, K. Guo, Q. Tang, Z. Zhang, H. Cheng, and Z. Zhao, "Leopard: Towards high throughput-preserving BFT for large-scale systems," in *Proc. IEEE 42nd Int. Conf. Distrib. Comput. Syst.*, 2022, pp. 157–167.
- [15] K. Mu, Y. Bo, A. Asheralieva, and X. Wei, "Separation is good: A faster order-fairness byzantine consensus," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2024.
- [16] M. Biely, Z. Milosevic, N. Santos, and A. Schiper, "S-Paxos: Offloading the leader for high throughput state machine replication," in *Proc. IEEE 31st Symp. Reliable Distrib. Syst.*, 2012, pp. 111–120.
- [17] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "An overview of blockchain technology: Architecture, consensus, and future trends," in *Proc. IEEE Int. Congr. Big Data*, 2017, pp. 557–564.
- [18] P. Zheng et al., "Aeolus: Distributed execution of permissioned blockchain transactions via state sharding," *IEEE Trans. Ind. Informat.*, vol. 18, no. 12, pp. 9227–9238, Dec. 2022.
- [19] C. Jin, S. Pang, X. Qi, Z. Zhang, and A. Zhou, "A high performance concurrency protocol for smart contracts of permissioned blockchain," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 11, pp. 5070–5083, Nov. 2022.
- [20] J. Liu, P. Li, R. Cheng, N. Asokan, and D. Song, "Parallel and asynchronous smart contract execution," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 5, pp. 1097–1108, May 2022.
- [21] J. Xu, C. Wang, and X. Jia, "A survey of blockchain consensus protocols," *ACM Comput. Surv.*, vol. 55, no. 13s, pp. 1–35, 2023.
- [22] S. Delgado-Segura et al., "TxProbe: Discovering Bitcoin's network topology using orphan transactions," in *Proc. Financial Cryptogr. Data Secur. 23rd Int. Conf.*, 2019, 23, pp. 550–566.
- [23] C. Decker and R. Wattenhofer, "Information propagation in the bitcoin network," in *Proc. IEEE P2P*, 2013, pp. 1–10.
- [24] T. Wang, C. Zhao, Q. Yang, S. Zhang, and S. C. Liew, "Ethna: Analyzing the underlying peer-to-peer network of ethereum blockchain," *IEEE Trans. Netw. Sci. Eng.*, vol. 8, no. 3, pp. 2131–2146, Jul.–Sep. 2021.
- [25] A. Miller et al., "Discovering Bitcoin's public topology and influential nodes," 2015. [Online]. Available: <https://www.cs.umd.edu/projects/coinscope/coinscope.pdf>
- [26] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, pp. 288–323, 1988.
- [27] relab/hotstuff, 2022. [Online]. Available: <https://github.com/rehab/hotstuff>
- [28] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proc. 29th Annu. ACM Symp. Theory Comput.*, 1997, pp. 654–663.
- [29] M. Xiang, Y. Jiang, Z. Xia, and C. Huang, "Consistent hashing with bounded loads and virtual nodes-based load balancing strategy for proxy cache cluster," *Cluster Comput.*, vol. 23, pp. 3139–3155, 2020.
- [30] A. Wahrstätter et al., "Blockchain censorship," in *Proc. ACM Web Conf. 2024*, 2024, pp. 1632–1643.
- [31] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. Hoboken, NJ, USA: Wiley, 2007.
- [32] Y. Chen et al., "Fore-runner: Constraint-based speculative transaction execution for ethereum," in *Proc. ACM SIGOPS 28th Symp. Oper. Syst. Princ.*, 2021, pp. 570–587.
- [33] H. Sukhwani, N. Wang, K. S. Trivedi, and A. Rindos, "Performance modeling of hyperledger fabric (permissioned blockchain network)," in *Proc. IEEE 17th Int. Symp. Netw. Comput. Appl.*, 2018, pp. 1–8.
- [34] P. Zheng, Z. Zheng, J. Wu, and H.-N. Dai, "XBlock-ETH: Extracting and exploring blockchain data from ethereum," *IEEE Open J. Comput. Soc.*, vol. 1, pp. 95–106, 2020.
- [35] S. Hemminger et al., "Network emulation with NetEm," in *Proc. Linux Conf. AU*, vol. 5, 2005, p. 2005.
- [36] J. Qi et al., "BIDL: A high-throughput, low-latency permissioned blockchain framework for datacenter networks," in *Proc. ACM SIGOPS 28th Symp. Oper. Syst. Princ.*, 2021, pp. 18–34.
- [37] L. Xu, Y. Chen, and K. Guo, "A distributed storage middleware based on Hbase and redis," in *Proc. Comput. Supported Cooperative Work Social Comput. 15th CCF Conf.*, 2021, pp. 364–380.
- [38] V. Saraph and M. Herlihy, "An empirical study of speculative concurrency in ethereum smart contracts," 2019, *arXiv:1901.01376*.
- [39] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin, "All about eve:execute-verify replication for multi-core servers," in *Proc. 10th USENIX Symp. Oper. Syst. Des. Implementation*, 2012, pp. 237–250.
- [40] M. Al-Bassam, "Lazyledger: A distributed data availability ledger with client-side smart contracts," 2019, *arXiv:1905.09274*.
- [41] L. Yang, S. J. Park, M. Alizadeh, S. Kannan, and D. Tse, "{DispersedLedger }:{ High-Throughput } byzantine consensus on variable bandwidth networks," in *Proc. 19th USENIX Symp. Netw. Syst. Des. Implementation*, 2022, pp. 493–512.
- [42] W. Yu, K. Luo, Y. Ding, G. You, and K. Hu, "A parallel smart contract model," in *Proc. Int. Conf. Mach. Learn. Mach. Intell.*, 2018, pp. 72–77.
- [43] S. Baheti, P. S. Anjana, S. Peri, and Y. Simmhan, "DiPETrans: A framework for distributed parallel execution of transactions of blocks in blockchains," *Concurrency Comput. Pract. Experience*, vol. 34, no. 10, 2022, Art. no. e6804.
- [44] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding concurrency to smart contracts," in *Proc. ACM Symp. Princ. Distrib. Comput.*, 2017, pp. 303–312.
- [45] A. Zhang and K. Zhang, "Enabling concurrency on smart contracts using multiversion ordering," in *Proc. Web Big Data 2nd Int. Joint Conf.*, 2018, pp. 425–439.
- [46] P. S. Anjana, S. Kumari, S. Peri, S. Rathor, and A. Somani, "An efficient framework for optimistic concurrent execution of smart contracts," in *Proc. 27th Euromicro Int. Conf. Parallel Distrib. Netw.-Based Process.*, 2019, pp. 83–92.
- [47] S. Pang, X. Qi, Z. Zhang, C. Jin, and A. Zhou, "Concurrency protocol aiming at high performance of execution and replay for smart contracts," 2019, *arXiv:1905.07169*.
- [48] J. Wang and H. Wang, "Monoxide: Scale out blockchains with asynchronous consensus zones," in *Proc. 16th USENIX Symp. Netw. Syst. Des. Implementation*, 2019, pp. 95–112.
- [49] J. Xu, Y. Ming, Z. Wu, C. Wang, and X. Jia, "X-shard: Optimistic cross-shard transaction processing for sharding-based blockchains," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 4, pp. 548–559, Apr. 2024.
- [50] L. Jia, Y. Liu, K. Wang, and Y. Sun, "Estuary: A low cross-shard blockchain sharding protocol based on state splitting," *IEEE Trans. Parallel Distrib. Syst.*, vol. 35, no. 3, pp. 405–420, Mar. 2024.
- [51] D. Hu, J. Wang, X. Liu, Q. Li, and K. Li, "LMChain: An efficient load-migratable beacon-based sharding blockchain system," *IEEE Trans. Comput.*, vol. 73, no. 9, pp. 2178–2191, Sep. 2024.
- [52] M. Li, W. Wang, and J. Zhang, "LB-Chain: Load-balanced and low-latency blockchain sharding via account migration," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 10, pp. 2797–2810, Oct. 2023.
- [53] J. Kwon and E. Buchman, "Cosmos whitepaper," *A Netw. Distrib. Ledgers*, vol. 27, pp. 1–32, 2019.
- [54] G. Wood, "Polkadot: Vision for a heterogeneous multi-chain framework," White Paper, 2016. [Online]. Available: <https://www.allcryptowhitepapers.com/wp-content/uploads/2019/08/PolkaDotPaper.pdf>



Shengjie Guan received the bachelor's degree in computer science and technology from the TaiShan College, Shandong University, Jinan, China, in 2021. He is currently working toward the PhD degree with the School of Computer Science, Peking University, Beijing, China. His main research interests include permissioned blockchain, consensus algorithm and sharding blockchain.



Rongkai Zhang received the bachelor's degree in computer science and technology from the School of Computer Science, Shandong University, Jinan, China, in 2023. He is currently working toward the PhD degree with the School of Computer Science, Peking University, Beijing, China. His main research interests include sharding blockchain and smart contract.



Jieyi Long received the bachelor's degree in microelectronics from Peking University, Beijing, China, and the PhD degree in computer engineering from Northwestern University, Evanston, IL. He is the co-founder and CTO of Theta Labs, Inc., a pioneer with the research and development of blockchain-based technologies. His current research interests include blockchain, generative AI, and various distributed computing topics.



Qiuyu Ding received the bachelor's degree in computer science and technology from the School of Computer Science, Beijing Institute of Technology, Beijing, China, in 2022. He is currently working toward the PhD degree with the School of Computer Science, Peking University, Beijing, China. His main research interests include blockchain and sharding blockchain.



Mingchao Wan is currently the Chief Architect of Blockchain with the Beijing Academy of Blockchain and Edge Computing. He is also a member of the Beijing Nova Program, 2021. Prior to his current role, he was a research scientist with IBM China Research Lab. His research interests include blockchain architecture, consensus and zero-knowledge proof.



Mingxuan Song received the bachelor's degree in computer science and technology from the School of Computer Science, China University of Geosciences, Wuhan, China, in 2023. He is currently working toward the PhD degree with the School of Computer Science, Peking University, Beijing, China. His main research interests include reinforcement learning (RL), sharding blockchain, and large language models (LLMs).



Taifu Yuan received the bachelor's degree in electronic information engineering from the School of Electronic Information, University of Science and Technology Liaoning, Anshan City, China, in 2014. He was a senior software development engineer with the Beijing Academy of Blockchain and Edge Computing. His research interests include blockchain and consensus.



Zhen Xiao (Senior Member, IEEE) received the PhD degree from Cornell University, Ithaca, NY, 2001. After that he joined as a senior technical staff member with AT&T Labs—Research, NJ, and then a research staff member with IBM T.J. Watson Research Center. He is currently a professor with the School of Computer Science, Peking University, Beijing, China. His research interests include blockchain, AI, and various distributed systems topics. He is a senior member of the ACM and IEEE.



Jin Dong is currently the general director with the Beijing Academy of Blockchain and Edge Computing. He was the director with Beijing Advanced Innovation Center for Future Blockchain and Privacy Computing. The team he leads has developed “ChainMaker”, the first hardware-software integrated blockchain system in the world. His research interests include blockchain, artificial intelligence, and low-power chip design.