Distributed and Parallel Databases (2024) 42:143–178
https://doi.org/10.1007/s10619-022-07409-7

An SGX-based execution framework for smart contracts upon permissioned blockchain

Min Fang[1] · Zhao Zhang[1,2] · Cheqing Jin[1] · Aoying Zhou[1]

Accepted: 8 April 2022 / Published online: 23 May 2022

Abstract

Since consensus protocol and execution mechanism act as two key factors for the overall throughput of blockchain systems, how to execute smart contracts efficiently becomes an emergent bottleneck as many high-performance consensus protocols have been proposed in recent years. Due to the existence of Byzantine nodes, existing concurrency approaches can only achieve intra-node concurrency, not inter-node concurrency. Fortunately, since the trust among nodes can be achieved based on the confidentiality guarantee provided by the trusted execution environment, such as Intel Software Guard Extensions (SGX), we propose a novel concurrent execution framework using SGX, which is the first to achieve both intra- and inter-node concurrency. Specifically, each replica executes the task assigned by the primary in parallel and gets trusted results using SGX firstly. Then, each node obtains the execution results of others via state replication to achieve consistency. However, we must ensure the integrity and correctness of all data transferred to SGX for getting the trusted results. Therefore, we design a novel approach to efficiently generate Merkle multiproofs and verify data in parallel. Theoretical analysis and experimental results show that the proposed scheme significantly outperforms state-of-art solutions.

Keywords Blockchain · Smart contract · Concurrency · SGX

✉ Cheqing Jin
cqjin@dase.ecnu.edu.cn

Min Fang
mfang@stu.ecnu.edu.cn

Zhao Zhang
zhzhang@dase.ecnu.edu.cn

Aoying Zhou
ayzhou@dase.ecnu.edu.cn

[1] School of Data Science and Engineering, East China Normal University, Shanghai, China

[2] Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, Guilin, China

Mathematics Subject Classification MSC code1 · MSC code2 · More

1 Introduction

As a decentralized, tamper-proof, traceable, and trusted distributed ledger jointly maintained by multiple mutual distrust participants, blockchain has shown broad application prospects and attracted a lot of attention from academics and industry since its inception [1–3]. The main factors that affect the throughput of blockchains are the efficiency of consensus protocol and smart contract execution mechanism. Some recent works focus on studying scalable consensus protocols, which attempt to improve throughput by designing or modifying consensus algorithms [2, 4–7]. However, with significant improvement of the consensus efficiency in the permissioned blockchain, how to execute smart contracts efficiently becomes a big challenge nowadays [8, 9]. Concurrency is a direct way to improve the performance of smart contracts. Existing solutions usually follow a two-phase style [8, 10–12], where the primary adopts a serializable concurrency control protocol to execute all transactions per block and generates scheduling logs in the first phase, then each replica replays all transactions according to the scheduling logs and verifies results in parallel in the second phase.

However, in a hostile environment, malicious nodes may deliberately discard or tamper with data, so a node distrusts any information from others. Under this situation, once a node receives a block, it needs to re-execute all transactions in the block to get the result. By transferring concurrent scheduling logs, the two-phase methods above improve execution efficiency to some extent, but the following two weaknesses still remain: (i) Both phases are still serialized, i.e., all replicas are idle when the primary is working; (ii) All replicas must replay all transactions to keep consistent. Fortunately, trusted hardware that has evolved deeply can deal with this issue. The trusted execution environment (TEE) such as Intel Software Guard Extensions (SGX) [13] provides an encrypted region, also known as *enclave*, to enable confidentiality and integrity protection of code and data inside it from other processes (privileged software, etc.) and physical attacks. The communication between SGX-equipped nodes is implemented through a particular secure channel, which is a remote-attestation transmission channel of SGX and can avoid data being maliciously tampered with. In other words, once the mutual trust has been established using SGX, smart contracts running in SGX are protected against threats from other processes. Consequently, consistency among nodes can be achieved via state replication without re-execution.

Therefore, we design a novel SGX-empowered framework to improve smart contract execution efficiency. At first, we use enclave to protect the execution result from attacks in the first phase (*Execution* phase). Then, the execution result is synchronized among nodes via state replication in the second phase (*Follow* phase). Hence, it is unnecessary to re-execute any smart contract by other nodes to verify data integrity. Moreover, smart contracts can be dispatched to different nodes and executed with a serializable concurrency control protocol, avoiding idle waite of replicas when primary is executing. Note that cross-contract calls

are identified in the *Execution* phase and processed in a deterministic order in the *Follow* phase. Therefore, the transfer cost between enclave and untrusted memory should be minimized in the *Execution* phase.

However, executing contracts directly in the enclave in the first phase does not significantly improve the efficiency due to the following reasons: (i) The enclave page cache (EPC) is limited to 128 MB (or 256 MB in the latest implementation), of which the available size is smaller than this [14, 15], it is infeasible to maintain all state data in the enclave if data volume exceeds the EPC limitation. Furthermore, it is hard to predict which part of state data will be visited by transactions before running because smart contracts are usually written in Turing-complete languages; (ii) The cost of transitioning control to/from enclaves, i.e., Ecall/Ocall, is high, resembling a context switch [16, 17]. Besides, enclave paging is expensive, bringing additional enclave transitions and cryptographic operations [18, 19]. It is not hard to devise an asynchronous execution method based on existing two-phase framework to deal with this issue. At first, a batch of transactions is divided into multiple micro-batches, each being pre-executed in untrusted memory to generate scheduling logs and collect visited data, and then validated in the enclave.

As a result, the aforementioned method suffers from three *challenges*: (i) How to minimize the number of enclave transitions and the amount of data transferred when the code and accessed state data of smart contracts exceed the storage limitation of EPC; (ii) How to improve validation efficiency of the enclave using pre-execution in the untrusted memory; (iii) How to verify the correctness for state data passed to the enclave from untrusted memory quickly. For the first challenge, we compress state data transmitted and cache part of hot data in the enclave to reduce the amount and number of data transfers. Furthermore, to minimize the transfer cost while maintaining execution efficiency, we use pipeline for data transmission between untrusted and trusted regions. Then, we propose a mixed concurrent control protocol optimized for traditional two-phase execution, which combines the batching *OCC* (Optimistic Concurrency Control) with traditional *OCC* to optimize the execution process. Meanwhile, we use Merkle proof [20] to ensure the integrity and correctness of state data passed into the enclave. For fast proof and verify data, we design a method to quickly get no-index multiproofs for multiple values based on Merkle Tree (*MT*) [21] and verify data quickly. According to the characteristics of the data access pattern, we further optimize the structure of *MT* and design a compressed *MT* to merge hot data. Moreover, a concurrent *MT* (*CMT*) is designed to support concurrent operations on a tree and is friendly to a large dataset.

The main contributions are summarized below:

- We propose a novel SGX-based two-phase concurrent approach to achieve both intra- and inter-node concurrency for smart contracts. Different contracts will be dispatched to different nodes to execute, the results will be verified in the enclave and synchronized via the secure channel. Cross-contract calls are also supported here.
- We propose a mixed *OCC* protocol to improve execution and validation efficiency, and design a *MT*-based mechanism to efficiently obtain compact Merkle

multiproofs and verify in parallel. Moreover, a compressed *MT* based on data access pattern is proposed to optimize it further.
– We design and implement a multi-threaded prototype integrated with the above technologies in open-source BFT-SMaRt [22] and evaluate it using the standard benchmark in a distributed setting. Experimental results show the efficiency of the proposed method.

As the extension of our conference version [23], we make the following improvements: (i) Add the process of *Follow* phase and solve the problem of cross-contract calls; (ii) Propose a mixed concurrent control protocol and a compressed *MT*; and (iii) Conduct reliable theoretical analysis and details of design.

1.1 Organization

The paper is organized as follows: Sect. 2 provides background about smart contracts and Intel SGX. Section 3 describes the overall architecture of our system model. Section 4 explains our SGX-based execution framework in detail. The relevant content of Merkle multiproofs is discussed in Sect. 5. Section 6 shows how to use the framework with PBFT and provides some performance and security analysis. The experimental evaluations are presented in Sect. 7. Section 8 reviews the related work. Finally, a conclusion is provided in Sect. 9.

2 Background

In this section, we present relevant background on smart contracts and SGX.

2.1 Smart contract

The smart contract concept was first introduced defined by computer scientist, lawyer, and cryptographer Nick Szabo in 1994 [24]. A smart contract is initially a computer protocol that disseminates, validates, or enforces a contract in an informative manner, allowing trusted transactions to be made without a third party, which are traceable and irreversible. Modern blockchain systems follow this idea and support a set of programmable smart contract implementations on top of cryptocurrency transactions, such as contract account in Ethereum [25] and Chaincode in Hyperledger Fabric [26], to better integrate into traditional industries. A smart contract, written in Turing-complete languages (e.g., Solidity [27]), can be viewed as a piece of code predefined by business logic that executes automatically triggered by a transaction. Figure 1 is part of the source code for a smart contract, which implements a simple payroll and bank application written in Solidity language. Generally, a smart contract can be called directly or indirectly. For example, in Ethereum, there are two types of accounts, namely normal accounts and contract accounts. When a transaction occurs between normal accounts, it is similar to a bank transfer transaction, such as the *SimpleBank* contract in Fig. 1. However, the situation is more

```
contract SimpleCompany {
    address[] private staffs;
    // more state definitions

    // pay salaries to all staff
    function deliverSalary (address bankAddress, uint256 salary) public {
        for (uint i=0; i<staffs.length; i++) {
            address staff = staffs[i];
            // call SimpleBank contract for paying salaries to a specific staff
            SimpleBank(bankAddress).sendPayment(msg.sender, staff, salary);
        }
    }
    // more operation definitions: addStaff, deleteStaff, updateStaff...
}

contract SimpleBank {
    mapping(address => uint256) private balances;
    // more state definitions

    // transfer money between two accounts
    function sendPayment (address addr1, address addr2, uint256 amount)
                                            public payable returns(bool) {
        if (balances [addr1]>=amount) {
            balances[addr1] = balances[addr1] - amount;
            balances[addr2] = balances[addr2] + amount;
            return true;
        }else{
            return false;
        }
    }
    // more operation definitions: depositChecking, amalgamate, writeCheck...
}
```

Fig. 1 A typical smart contract example

complicated when the transaction involves a contract object. In this situation, this transaction may be a transfer, a call to a contract function, or both. This is the case with the *SimpleBank* contract at the top of Fig. 1, where contract function *deliverSalary* invokes *SimpleBank* contract and pays salaries to a specific staff by function *sendPayment*. Under this case, if a transaction calls the function *deliverSalary* of *SimpleCompany* contract, it will trigger a call to the *SimpleBank* contract, then we call it a cross-contract transaction.

2.2 Intel SGX

The trusted execution environment (TEE) is a secure area within CPU, which can guarantee the confidentiality and integrity of computation and data in it. As a state-of-the-art implementation of TEE, Intel Software Guard Extensions (SGX) has attracted broad attention from academia and industry since its inception. SGX is an extension of x86 instruction set architecture that enables applications to run inside an isolated virtual address space called enclave [13]. Both code and data of enclave reside in a protected area of physical memory called the enclave page cache (EPC) that is not accessible to the host, i.e., the rest of processes outside the enclave. The

Memory Encryption Engine (MEE) is responsible for encrypting and decrypting the cache line in the EPC, and the code and data do not leave enclave memory with unencrypted. SGX provides a remote attestation mechanism that allows a remote party to verify the authenticity of the enclave's identity information and the code/data it loads. It facilitates the creation of a secure channel between the remote party and the enclave (e.g., passing secret keys).

SGX can build trusted and secure execution environments for untrustworthy blockchains. However, due to the limitations of SGX implementation, the following key challenges still remain to achieve a robust and high-performance design. (i) The preserved memory space for SGX is limited to 128 MB (or 256 MB in the current implementation). Although SGX supports swapping unused EPC pages to unprotected memory to hide this limitation, the overhead of page swapping is high, which can reach 40,000 CPU cycles [14, 19]. (ii) The cost of transitioning control to/from enclaves (i.e., Ecall and Ocall) is high, resembling a context switch, which can easily consume 8600–14,000 CPU cycles according to cache hit or miss [19, 28]. And a page missing will bring the cost for enclave transitions and extra computation of cryptographic operations. (iii) SGX faces security risks such as excessive TCB and exposure of excessive external interfaces, because any vulnerabilities in the loaded code can endanger the security of the entire system. Some works attempt to deploy function libraries or system libraries in the enclave to support the execution of a complete application, thereby reducing the interaction between the code in the enclave and the operating system [14, 15]. These works support unmodified applications and can reduce the attack surface while ensuring a small TCB. Among the above three limitations, the first two focus on performance, while the last one is for security.

3 Overview

In this section, we present the overview of our novel SGX-based concurrent execution framework, including system architecture and adversary model.

3.1 System architecture

We extend concurrency to improve smart contract process with the confidentiality guarantee provided by SGX, achieving both inter- and intra-node concurrency. Specifically, a batch of transactions will be assigned based on some partition rules to multiple nodes for concurrent execution using SGX, which will be synchronized to others via Gossip protocol. Figure 2 elucidates our system architecture that involves: (i) One primary (*PrimaryNode*), a node elected by consensus, which is responsible for packaging and broadcasting blocks. (ii) Multiple replicas (*ReplicaNode*s), nodes other than primary, which can be further divided into *ExecutionNode* and *FollowNode* according to logical roles, and the role of each node is different for the same contract. Each *ReplicaNode* is responsible for the execution, validation, and

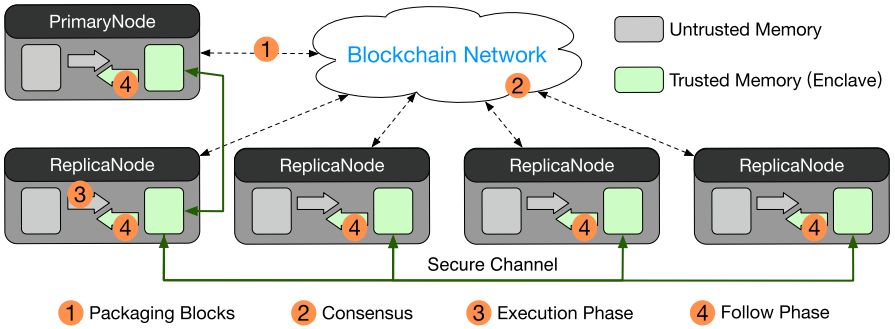


Fig. 2 System architecture

maintenance of its contract state. All nodes are SGX-equipped and have trusted and untrusted memory.

A batch of smart contracts goes through four steps from packaging to committing, as shown in Fig. 2.

① *PrimaryNode* packages a batch of transactions into a block and assigns the transactions to different *ReplicaNode*s according to some pre-specified rules, which are used to divide a batch of transactions rationally and can take various forms, like geo-distributed-based [29] and contract-based [30], etc. The allocation information is attached to the block body and participate in consensus along with the block (see Sect. 4.1 for details).

② *PrimaryNode* initiates a consensus process and each *ReplicaNode* acts as an *ExecutionNode* for the assigned transactions.

③ *ExecutionNode* executes pre-assigned transactions concurrently, marks and aborts all cross-contract transactions during execution. Meanwhile, *ExecutionNode* uses enclave to get trusted execution results (see Sect. 4.2 for details).

④ All results and cross-contract transactions are transferred to *FollowNode*s via the secure channel of SGX. *FollowNode*s synchronize results via state replication and re-execute all unexecuted transactions with the same sequence obtained in the consensus (see Sect. 4.3 for details).

3.2 Adversary model

In permissioned blockchain, all nodes are considered potential adversaries because no one trusts others. To avoid any replica re-executing all smart contracts, we apply SGX to guarantee the correctness and integrity of execution results. Since the size of SGX is limited, all necessary data in the enclave for validating are from untrusted memory. Here, we adopt Merkle proof [20] to guarantee the correctness and completeness of the initial data passed into the trusted memory, because the probability of verification based on the wrong initial state or proof is negligible [31]. And *FollowNode* can trust the correctness and completeness of the encrypted result because an adversary cannot break the hardware security enforcement of SGX even though

it may compromise OS or other privileged softwares. SGX side-channel, page-table, and cache-timing attacks are beyond the scope of this work because these vulnerabilities are specific implementations and common to all SGX frameworks [32–34].

4 The SGX-based execution framework

To improve smart contract execution efficiency, we design a novel framework based on SGX. In this section, we present our execution framework in detail, including the packaging strategy, the *Execution* phase and the *Follow* phase.

4.1 The packaging strategy

Randomized packaging mechanism is widely used in the existing permissioned blockchain, where a specific thread randomly packages a batch of transactions into a block. This packaging mechanism brings low system latency because it obtains transactions from transaction pool without the division of transactions. However, such randomized approach cannot restrict the number of distinct contracts in a block. For example, when there are t transactions in a block, in the worst case, these t transactions all call the same contract. In this situation, if we simply assign transactions in a block to different nodes for execution, more cross-contract transactions may occur, because there may be more possible conflicting dependencies among transactions for the same contract [30]. Meanwhile, the cost of processing cross-contract transactions is high. However, considering user rights and data security, the conflicting dependence between transactions of different contracts may be smaller [35, 36]. Therefore, if there are m contracts and the number of transactions per contract in a block is $\frac{t}{m}$, we can divide this block into $\frac{t}{m}$ equal parts and execute them in parallel by different nodes.

Therefore, in order to improve efficiency and achieve load balancing, we maintain multiple queues in *PrimaryNode*, responsible for packaging and distributing transactions. The number of queues is equal to the number of smart contracts. As shown in Fig. 3, each queue (called contract queue) maintains transactions that call to the same contract. Each block can be composed of one or more *contract block*s, and

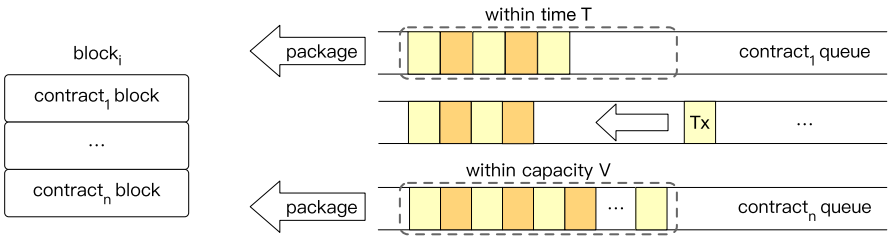


Fig. 3 The packaging strategy

each *contract block* is composed of transactions in the same contract queue. Like Ethereum transactions, the recipient address of a transaction is usually specified, so we can know which contract is invoked by analyzing the content of the transaction. Whenever receiving a transaction, *PrimaryNode* puts it into the corresponding queue based on the information obtained about the recipient address of the transaction. The cost of this is minimal, as it is easy to read the value of a specified field of a transaction directly. Besides, we can apply multiple threads to facilitate this process, just like the multi-threaded deep pipeline designed in [37]. Then, *PrimaryNode* packages *contract block*s according to some rules to get a block, and transactions of different contract types are packaged into different *contract block*s. We design two strategies to implement packaging. One is based on time, for each contract queue, after a fixed period T, even if it is unfull, *PrimaryNode* will still package all transactions in it. Another is based on capacity, when the number of transactions in a contract queue exceeds pre-defined threshold V, *PrimaryNode* will package all transactions in each contract queue and generate multiple *contract block*s. All *contract block*s will be packed into a block to participate in consensus. It is not difficult to find that the system latency will increase under a great T, because there may be some unprocessed transactions in the queue. To simplify the description, we denote the number of transactions contained in a block as the block size. Therefore, if V is small, the block size also small since at most $V \cdot m$ transactions are packaged into a block. According to the experimental results in Fig. 15 in Sect. 7.3, a smaller block will lower the throughput. Therefore, we can optimize this packaging process using a shorter period T and a greater queue capacity V. When a block is generated, *PrimaryNode* specifies a particular *ReplicaNode* as *ExecutionNode* for each *contract block* in this block, i.e., idle replicas with no or few tasks, and broadcasts this block.

4.2 Execution phase

The upper part of Fig. 4 shows the *Execution* phase in an *ExecutionNode*, and the bottom part illustrates the *Follow* phase in a *FollowNode*, where they communicate through the secure channel of SGX. During the *Execution* phase, we pre-execute transactions concurrently in untrusted memory and then concurrently validate the correctness in trusted enclave, as shown in the upper part of Fig. 4. All state data accessed by the assigned tasks ($contract_j$ *block*) are identified during pre-execution and passed to the enclave in batches to minimize the number of enclave transitions. At the same time, the proofs of all identified data are also transmitted to the enclave to provide integrity assurance. How to efficiently obtain proofs and verify data with proofs are discussed in Sect. 5.

4.2.1 Pre-execution in the untrusted memory

Conflict graph (CG) has long been used in concurrency control to capture conflicts between transactions [8, 38, 39], where vertices are transactions, and edges represent the dependencies of *read-write* conflicts. A *CG* can be generated based

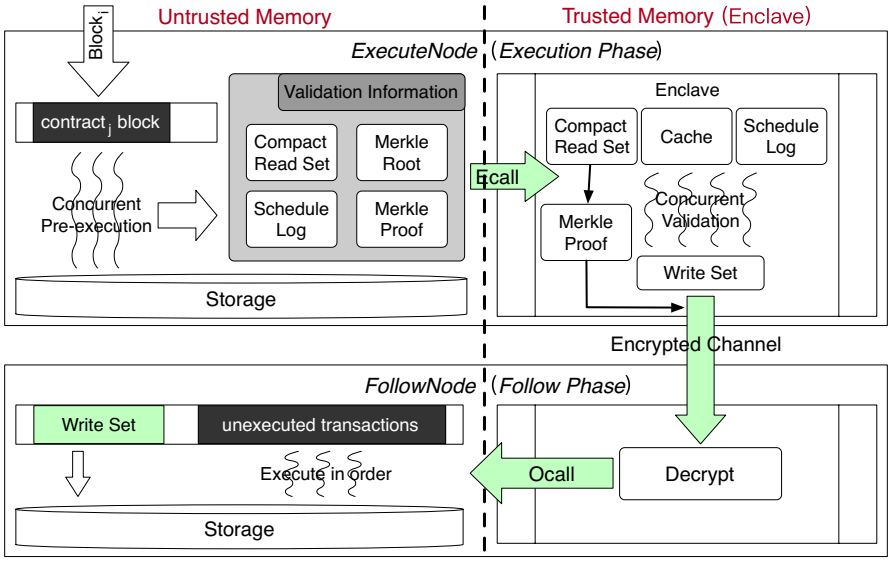


Fig. 4 The *Execution* phase and the *Follow* phase

on the read-write sets of transactions after concurrent execution and partitioned into several non-intersecting subgraphs, so transactions in different subgraph can be executed in parallel safely. In this paper, we also construct *CG* to represent the dependencies between transactions. Further, we define $RS(T_i)$/$WS(T_i)$ as the read/write set of a transaction T_i, and a dependency $T_j \rightarrow T_i$ $(i \neq j)$ exists if $RS(T_j) \cap WS(T_i) \neq \emptyset$. With the scheduling logs produced by non-intersecting subgraphs, the transactions can be executed concurrently and deterministically in the enclave. Since each block contains a batch of transactions in the blockchain, some recent works apply batching *OCC* protocol to execute smart contracts and use reordering mechanism to improve the commit rate [8, 39]. Basically, such methods contain two steps. First, a batch of transactions is processed based on the current snapshot, and a *CG* is constructed based on the read/write set obtained after execution. Second, conflicting transactions are removed from CG to ensure the correctness of results, and the schedule log is generated.

Example 1 Consider three transactions T_{1-3} in Table 1. Operations $r_i(D)$, $w_i(D)$ and $rw_i(D)$ denote the read, write, and read and write data item D of transaction T_i, respectively. Operations c_i and a_i denote the i-th transaction prepares to commit and abort, respectively. Under the traditional OCC protocol, once T_1 is successfully committed, T_2 and T_3 must be aborted because data item A has been updated. However, the situation may differ under batching OCC. Figure 5a shows the conflict graph of transactions T_{1-3} under batching OCC. In order to get the serializable order, i.e., topological order, we either abort T_1 or T_3 to break the circle (T_3 is aborted as an

Table 1 Scheduling information

Transaction	T_1	T_2	T_3	T_4
Scheduling				$r_4(A);w_4(B)$
				c_4
	$r_1(B);rw_1(A)$	$r_2(A)$	$r_3(B);rw_3(A)$	
	c_1			
		a_2	a_3	

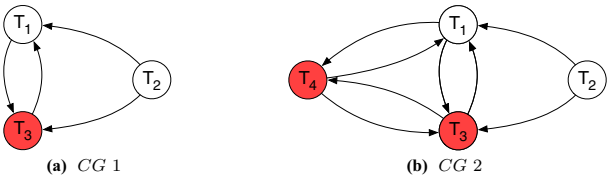


Fig. 5 Examples of *CG*, red vertices are aborted for serialization (Color figure online)

example). Therefore, we can commit two transactions T_2, T_1 in order successfully rather than one under traditional OCC protocol.

Note that the rollback rate of batching *OCC* will increase significantly under high conflict workload. Furthermore, batching *OCC* behaves even worse if conflict-free transactions have not been committed early. For example, if we execute T_{1-4} in Table 1, T_4 can be committed in traditional *OCC* but must be aborted in the batching *OCC* because of new circles (T_1 and T_4 form a circle; T_3 and T_4 form a circle). Obviously, for a large batch of transactions, execution under batch *OCC* may have more unresolved conflicts, in other words, more circle dependencies. This is comparable to the observation of Hyperledger Fabric, where the number of transactions included in one block has significant effect on the number of transaction failures [40]. Based on this, we combine the advantages of traditional *OCC* and batching *OCC*. We identify committable transactions during execution, which can reduce the possibility of additional conflicts and the cost of resolving conflicts. For this purpose, we introduce *Rule* 1 below.

Rule 1. A transaction T_i can be early-committed if there is no dependency for each uncommitted transaction T_j.

Rule 1 is correct because there is no dependency between a commitable transaction and any others. In addition to early-commit, we should also identify all unresolvable conflicts and abort these transactions, especially under high conflict workloads, i.e., break unresolvable circle dependencies as soon as possible. However, such dependencies are difficult to detect during execution due to multiple

transactions or data items may be involved. Since a circle dependency between two transactions can be determined easily, we introduce *Rule* 2 to abort uncommittable transactions as soon as possible.

Rule 2. A transaction T_i must early-abort if there are two dependencies $T_i \rightarrow T_j$ and $T_j \rightarrow T_i$ for an uncommitted transaction T_j.

Rule 2 is valid due to the existence of a circle dependency between two transactions. Observe that for a long transaction, the read/write sets may be large. Therefore, in order to evaluate *Rule* 2 efficiently, we restrict it to a concrete case where two dependencies must be triggered by the same value, such as T_1 and T_3 in Example 1 (dependency is triggered by the same value A). Such a situation is common in the blockchain because, in general, smart contract access to accounts is always skewed [41, 42], namely, some accounts are frequently accessed, also known as active accounts.

We can optimize the pre-execution process under high conflict workload based on *Rule* 1 and 2. Specifically, following the standard *OCC* protocol, we divide the execution process of each transaction into three phases, including local execution, validation, and early-commit. However, during validation, we only commit transactions that satisfy *Rule* 1 and rollback all that meet *Rule* 2. After executing the transactions in current batch, we use the method in [8] to resolve the conflicts to obtain scheduling logs.

Example 2 Consider four transactions T_{1-4} in Table 1. According to *Rule* 1, T_4 can be early-committed because there exists no dependency between T_4 and any other transactions when it comes to the validation phase. Subsequently, T_{1-3} begin to be executed and validated. Noted that either T_1 or T_3 needs to be aborted under *Rule* 2 because they try to update the same item A. Same as Example 1, we also abort T_3. However, T_1 still cannot be early-committed, because uncommitted T_2 has already read A. Since T_2 is a read-only transaction, it can be early-committed according to *Rule* 1. After execution, there is no circle in CG because only T_1 is left, and we can commit it directly. As a result, we mitigate the cost of resolving conflict dependencies in [8]. In addition, we can commit three transactions T_4, T_2 and T_1 in order successfully rather than two in Fig. 5b.

4.2.2 Validation in the enclave

Due to the size limit of enclave and the expensive cost of enclave paging, we get all necessary state data for execution during pre-execution phase and compress it further to generate a compact read set, i.e., the oldest read operations for the current batch of transactions. After transferring the compact read set into enclave, transactions will be re-executed based on the incoming read set. However, it introduces a certain waiting time because pre-execution in untrusted memory and re-execution

in trusted memory are synchronous. Therefore, we transfer data in batch asynchronously to connect untrusted and trusted memory to optimize parallelism. In the untrusted memory, we pre-execute transactions in micro-batch, e.g., we divide b transactions into c micro-batches and execute these batches sequentially. Under this situation, we must ensure the size of $\frac{b}{c}$ transactions are relatively reasonable to maximize the transfer volume without adding an additional high transfer overhead, which can be approximated by the Intel VTuneTM XE Amplifier [18]. Since an EPC page size is 4 kB [43], which is used as the default granularity in some works to optimize enclave transfers [44], so we set the default value of $\frac{b}{c}$ to 4 kB. Transactions in each micro-batch are executed using the method mentioned above. During execution, we collect all aborted transactions and put after the last micro-batch to execute again since the transactions aborted in the current micro-batch have high probability of being aborted again. In addition, data transmission of the current micro-batch can be performed in parallel with the pre-execution of the next micro-batch.

In order to minimize the amount of data transferred, we cache the latest state data involved in the previous micro-batch in the enclave. Meanwhile, a corresponding buffer is maintained in the untrusted memory to identify the cached data items. Since the enclave size is limited, we use the LRU page replacement algorithm to manage the cache dynamically. Considering the existence of Byzantine nodes, we must provide proofs for transferred data and validate in the enclave to ensure correctness (see in Sect. 5 for the acquisition and verification of proofs). Thanks to the confidentiality and integrity protection provided by SGX, there is no need to provide proof for data inside the enclave, so the transmission cost is reduced significantly. During validation, the compact read set is validated using the Merkle root and its corresponding Merkle proof. After validation, transactions will be concurrently replayed based on transferred scheduling logs and the compact read set to get write set. Finally, a certificate is generated to show the correction of the execution result and passed to other nodes via the secure channel of SGX.

4.3 Follow phase

After the *Execution* phase, the trusted result obtained in the enclave is synchronized to *FollowNode* through the secure channel of SGX with Gossip protocol, and then the *Follow* phase begins. This subsection presents the entire *Follow* phase, including data synchronization and the execution of cross-contract transactions.

4.3.1 Synchronization process

As shown at the bottom of Fig. 4, after receiving the result, *FollowNode* will decrypt and store the information locally. As described in Sect. 3.1, all unexecuted transactions will be executed in the same order by all nodes after consensus. To avoid inconsistency of system state when partial results are missed, we require *FollowNode* to receive all results from others before updating state. But this naive method undoubtedly adds waiting time. Moreover, in a hostile environment, malicious nodes may deliberately discard or tamper with data. Although all malicious

actions can be discovered during validation in the enclave, data loss caused by hardware or network failures cannot be detected. Besides, the SGX of a node cannot determine what caused it not to receive results for an extended period, network failure or node failure, or both. Therefore, we maintain a timeout parameter inside the enclave to identify the state synchronization, which cannot be tampered with. The timeout parameter is initialized to the time used for one round of network communication during consensus and can be dynamically adjusted based on the time used for the last round of state synchronization. Once the result of a *contract block* is not received on time, SGX will send a confirmation message to the corresponding *ExecutionNode* via the secure channel. This confirms either *ExecutionNode* left the trusted network or just failed to execute all transactions on time due to some long transactions. For the former, it is no need to wait any longer, and the corresponding transaction will be marked as unexecuted. For the latter, SGX will reset the timeout parameter and continue to wait.

However, this method does not cope well with the case where nodes are deliberately unresponsive. To address this issue, we can delegate a task to multiple executors for execution with a deterministic method, i.e., *PrimaryNode* randomly selects g nodes at a time as an execution group for a task. In fact, for a cluster of $n = 3f + 1$ nodes (i.e., f malicious out of n nodes), the probability that there are only malicious nodes in an execution group is $p = \prod_{i=0}^{g} \frac{f-i}{3f+1-i}$. We allow the user to choose an upper bound of probability p to compute the appropriate g to trade off the probability of tolerating silence against the efficiency of concurrent execution. Specially, if we need to avoid the performance degradation caused by malicious node silence with 100% (i.e., $p = 0$), we must guarantee that each task has $g = f + 1 (> f)$ executors (i.e., at least one honest node). Although this strong guarantee improve system security, it significantly affects the concurrency efficiency of the proposed framework as only two contracts can be executed simultaneously. But when p is higher we can obtain better concurrency efficiency, e.g., when $f = 10$ and $g = 6$, i.e., $p < 0.1\%$, we can get 5 execution groups and execute 5 tasks simultaneously. Although this method reduces the inter-node concurrency of the system to some extent, it can bring higher security because as long as one executor isn't a Byzantine node and the system is not affected.

4.3.2 Cross-contract execution

A cross-contract transaction may have several sub-calls to other contracts, which will result in conflicts among transactions that invoke these contracts. Under this situation, some transactions must be aborted to keep atomistic, which limits the efficiency of concurrent execution among nodes. Therefore, we urgently need to resolve such cross-contract transactions. Generally, direct contract calls are usually specified, so contract call information can be obtained directly, just like Ethereum transactions. However, we cannot know which contract is called indirectly because indirect contract call is triggered called by the executed contract and can only determined by executing contract code. As shown in Fig. 1, the *SimpleBank* contract is triggered called by the function *deliverSalary* in the *SimpleCompany* contract. If

there are multiple contracts like *SimpleCompany* for different banks, we can't know which bank contract is called without executing. Hence, if a smart contract is called indirectly, we cannot obtain the contract information conveniently.

For these indirect contract calls, the current work [30] first pre-executes transactions in the client to identify all contracts involved and then executes the transactions related to these contracts together to avoid cross partition transactions during execution. Although this method does address the problem of executing cross-contract transactions, it puts an amount of computational pressure on clients, which is too heavy for light clients. Given this, we use a static code analysis tool Slither [45] for Solidity language to estimate the involved contracts for each function of a smart contract. In particular, Slither will over-approximate all possible contracts called by a transaction in all code paths. Based on the results obtained from analysis, *PrimaryNode* will merge the contract queue involved and then add the request to this merged queue. In addition, we adopt the merging strategy designed in [46] to achieve efficient queue merging. However, this method cannot suit for the dynamic contract calls where the contracts involved can only be discovered during execution. For this type of transaction, we abort them during execution and re-execute them after synchronizing the results. Specifically, during the *Execution* phase, an *ExecutionNode* finds all transactions with dynamic contract calls and aborts them to ensure consistency among nodes. Then the transactions ids are sent to other nodes via the secure channel of SGX. During the *Follow* phase, a *FollowNode* first verifies the correctness of the received result, and state synchronization will be performed once the verification succeeds.

After state synchronization, all nodes have consistent state. We need to ensure that all unexecuted transactions are executed deterministically on all nodes to keep this consistent. Serialization is the simplest way to achieve this, but it limits the concurrency capability of the system. In fact, for the same batch of transactions on the same data snapshot, their read and write states are also the same, which helps get the same *CG*. Given this, we implement a deterministic concurrent control protocol based on the mixed *OCC* we designed in Sect. 4.2.1. Therefore, as long as each node can abort the same transaction every time, the final scheduling order is the same.

Specifically, all cross-contract transactions are first executed locally. After executing, we construct a *CG* based on the read-write sets obtained during execution and break *CG* deterministically to eliminate conflicts. As shown in Fig. 6, Tx_{1-3} are cross-contract transactions, and all of them indirectly invoke another contract during execution, which triggers other transactions. For example, Tx_3 invokes contract S_2, and triggers transactions Tx_{31} and Tx_{32}, which are called internal transactions.

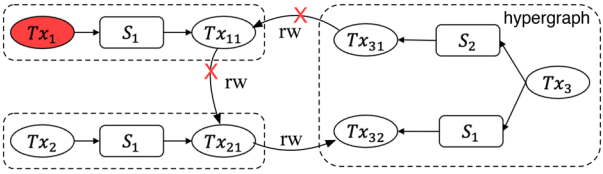


Fig. 6 The example of cross-contracts *CG*, red vertex is aborted for serialization (Color figure online)

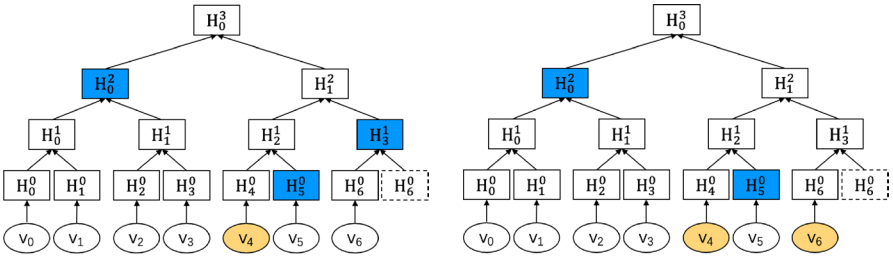
We define the cross-contract transaction and its all internal transactions as a *hypergraph*. The transactions within the *hypergraph* are executed in the invoking order. Since the scheduling of all internal transactions also satisfy ACID requirement, they must commit or roll back simultaneously. Finally, we convert it into the scheduling problem for individual *hypergraph*s, which can be dealt with by a similar solution in Sect. 4.2.1. For example, in Fig. 6, there are three *hypergraph*s marked with dashed boxes, based on the read-write dependencies of which a conflict graph *CG'* is formed. We first find all strongly connected *hypergraph*s of *CG'*, and then remove a *hypergraph* with the smallest transaction id to break the circle, instead of the random removal in [8]. Thus, we remove the first *hypergraph* produced by Tx_1, and the corresponding read-write dependencies are broken. Finally, we obtain a scheduling log with high parallelism, i.e., $Tx_2 \Rightarrow Tx_{21} \rightarrow Tx_3 \Rightarrow Tx_{31,32}$.

In summary, we avoid many cross-contract transactions through static analysis using Slither during packaging, and efficiently execute the remaining cross-contract transactions that are only identified during execution in the *Follow* phase.

5 Compact merkle multiproofs

MT is widely used to verify the integrity of data in blockchain [47, 48]. In general, a *MT* is a binary tree, which consists of some hash values. Let H_j^i (i is the level of the tree, j is the index of level i) denote a node in *MT*, which has two child nodes H_{2j}^{i-1} and H_{2j+1}^{i-1}. Figure 7a illustrates an example of *MT*. In our execution framework, we use *MT* to ensure the correctness of data transferred into the enclave. As SGX being memory-limited, the generated proof must be small enough. Thus, we consider the multi-value proof problem of *MT*, which tries to prove multi-values exist on the same *MT* at the same time. Although the existing methods on multiproofs [49–51] can save storage space to a certain extent, the following problems still exist: (i) Expensive to generate and verify proofs; (ii) Lack of scalability to support a large *MT*.

In fact, when generating the Merkle proof for an input value, it needs one proof value in each level of *MT* for the proof path. For example, in Fig. 7a, if we want to provide the proof for v_4, we need H_0^5 in level 0, H_3^1 in level 1, and H_0^2 in level



(a) The proof of one value **(b)** The multiproof of two values

Fig. 7 The typical example of Merkle multiproof

2 to form the proof path. When verifying, it calculates step by step from bottom to top according to the proof path until the root. However, when generating the proof for two values that belong to the same parent node, there is no need to add additional proofs. As shown in Fig. 7b, if we need to provide proof for v_4 and v_6, since H_3^1 can be calculated by v_6, there is no need to maintain it in the proof path to get H_1^2 of level 2. Given this, when obtaining proof for a verified set, we can determine whether two input values belong to the same parent node or not level by level to remove redundant intermediate node, such as H_3^1 in Fig. 7a. Next, we will explain how to efficiently obtain the multiproof and verify based on it.

5.1 Generate compact Merkle multiproofs

We design Algorithm 1 to obtain multiproofs of a *MT* efficiently. The inputs are a *MT H* and the ordered index set γ of the read set. Algorithm 1 iterates *H* from bottom to top and finally outputs a compact multiproofs $M=\{M^1, M^2, ...\}$, where M^i means the proofs of *i*-th level (Level-*i*) and consists of some hashcodes. *Proof* in Fig. 8 illustrates an example of *M*. Let *nd.lNode* and *nd.rNode* denote the left and right children of a node *nd*, respectively. If only *nd.lNode* exits in the input list, then *nd.rNode* must be output as a proof to obtain *nd*, and vice versa. If both

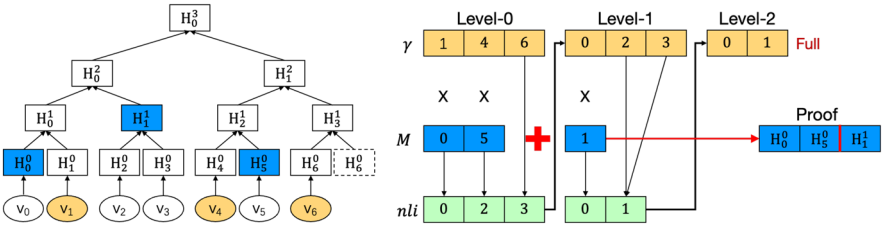


Fig. 8 An example of generating multiproofs

nd.lNode and *nd.rNode* are in the input list (called a full pair), it is no need to provide any proof to get *nd*.

Algorithm 1: Generate Compact Merkle Multiproofs

Input: the Merkle Hash Tree H, and the ordered index list γ of read set
Output: the compact Merkle multiproofs M

```
1  for each H^i in H do
2      if H^i.length = γ.length then
3          break
4      create the next level index list nli
5      for each index ω in γ do
6          if H^i_ω is an rNode then
7              M^i.add(H^i_{ω-1}) /* add left brother */
8          else if H^i_ω is an lNode then
9              if ω + 1 in γ then
10                 delete ω + 1 from γ (skip the right brother)
11             else if H^i_ω is not the last one of H^i then
12                 M^i.add(H^i_{ω+1}) /* add right brother */
13         nli.add(ω/2) /* maintain the parent index of ω */
14     γ ← nli
15 return M
```

Algorithm 1 generates proofs for each level in a bottom-up manner and finally generates compact multiproofs M. If all nodes in a level are in the input list γ, we do not need to generate proofs for higher levels (Lines 2–3). Otherwise, we must compute the proof for each item in γ and maintain their parent index at *nli*, which will be used as input to get M^{i+1} (Line 4). For each item ω in γ, if H^i_ω is an *rNode*, its left brother $H^i_{\omega-1}$ will be provided to get $H^{i+1}_{\omega/2}$, and maintained in M^i (Lines 6–7). If H^i_ω is an *lNode*, we will judge whether its right brother index $\omega+1$ is in γ or not. If it is, we just skip its right brother, i.e. delete ω from γ (Lines 9–10). Otherwise, we add $H^i_{\omega+1}$ to M^i if it is not the last one of H^i (Lines 11–12). Finally, we add the parent index of ω, i.e., $\frac{\omega}{2}$, to *nli* (Line 13). After current iteration, we replace γ with *nli* for Level-i+1. After iteration, no-index Merkle multiproofs M will be returned. Algorithm 1 dramatically reduces the transmission cost because there is no need to provide indexes and other additional information for multiproofs, which is necessary in previous works.

Example 3 As shown in Fig. 8, the read set that needs to provide proof is $\{v_1, v_4, v_6\}$, and the index list γ is initialized to the ordered index of this read set, namely $\{1, 4, 6\}$. We traverse γ in order. The first value is 1, an *rNode*, so its left brother H^0_0 used to get the parent H^1_0, and we add index 0 to the next index list *nli* for the next level. The second value is 4, an *lNode*, so it needs a right brother H^0_5 to get H^1_2, and add index 2 to *nli*. Since the last value 6 in γ is the last value without right neighbor, it will generate H^1_3 with itself, and add index 3 to *nli*. Finally, we get the proofs of Level-0, namely $M^0 = \{H^0_0, H^0_5\}$ and update $\gamma = nli = \{0, 2, 3\}$ for the next round. We iterate to get $M^1 = \{H^1_1\}$. At last, since γ in the Level-2 is full, namely the length

of γ is equal to the node number of in Level-2, Algorithm 1 is completed and returns a compact Merkle multiproof $M = \{\{H_0^0, H_5^0\}, \{H_1^1\}\}$.

5.2 Verify based on Merkle multiproofs

To reduce the amount of data transferred into the enclave, we produce an index-less multiproofs with Algorithm 1. Based on multiproofs obtained, we further propose Algorithm 2 to accelerate the verification process. The inputs of Algorithm 2 are the compact Merkle multiproofs M and the ordered node list β of read set, which consists of the index and hashcode of an input value, e.g., β of Level-0 in Fig. 9. The output of Algorithm 2 is a root obtained based on M, which will be compared with the root of H to validate the correctness of the transferred read set.

Algorithm 2: Verify Based on Merkle Multiproofs

Input: the compact Merkle multiproofs M, the ordered node list β of read set
Output: the computed root of M

```
while β.length>1 || M ≠ ∅ do
    create the next level node list nln
    for each node H_ω in β do
        create its parent node H_{ω/2}
        if H_ω is an rNode then
            H_{ω/2} ← hash(M^0.pop(), H_ω)
        else if H_ω is an lNode then
            if its right brother H_{ω+1} in β then
                H_{ω/2} ← hash(H_ω, H_{ω+1})
                delete H_{ω+1} from β
            else if M^0 ≠ ∅ then
                H_{ω/2} ← hash(H_ω, M^0.pop())
            else
                H_{ω/2} ← hash(H_ω, H_ω)
        nln.add(H_{ω/2})
    pop a tuple if M is not empty
    β ← nln
return β[0]
```

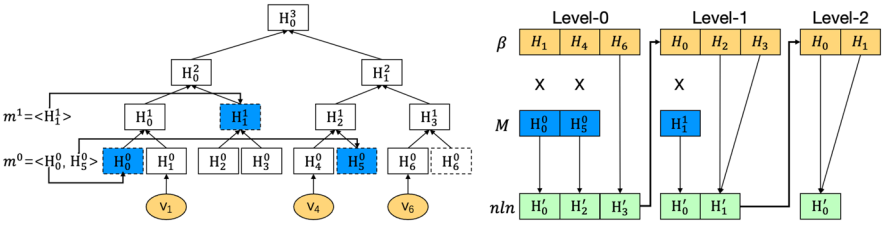


Fig. 9 An example of verification based on multiproofs

Algorithm 2 verifies the input data based on M in a bottom-up manner to get *Root*. If M is empty and the size of β is equal to 1, the root has been obtained and the process is ended (Line 1). For each iteration, we create the next node list *nln* firstly, which will be used as input for the next level verification (Line 2). Since *MT* is a binary tree, so the parent node of H_ω is $H_{\frac{\omega}{2}}$ (Line 4). For each node H_ω in β, if H_ω is an *rNode*, there must be a left brother in M^0, so we pop it from M^0 and compute its parent node with H_ω (Line 6). If H_ω is an *lNode* and the right brother $H_{\omega+1}$ is also in β, we compute $H_{\frac{\omega}{2}}$ with both of them and skip the right brother, i.e., delete $H_{\omega+1}$ from β (Lines 9–10). But if the pair is not in β, we should compute its parent node with a value in M^0 or compute with itself (Lines 11–14). Finally, we add the parent node $H_{\frac{\omega}{2}}$ into *nln* (Line 15). After dealing with the current level, if M is not empty, we pop the proof of the next level from it and update β to *nln* for the next level verification (Lines 16–17). By Algorithm 2, we return the last value in β as the *Root*.

Example 4 According to Example 3, we obtain multiproofs M={$\{H_0^0, H_5^0\}$, $\{H_1^1\}$} of given read set, and the ordered node list of read set $\beta = \{H_1, H_4, H_6\}$. As shown in Fig. 9, we traverse M in order based on Algorithm 2. Firstly, we use M^0 and β to verify and generate the node list of Level-1. Since the first node H_1 in β is an *rNode*, we compute H_0' by combining H_1 with H_0^0 in M^0, and put H_0' into *nln* as the inputted value of Level-1. The second node H_4 is an *lNode*, so we compute H_2' by using it and H_5^0 in M^0. When computing the last node H_6 in β, since both β and M^0 are empty, we get H_3' by computing H_6 with itself. Finally, we get the node list of Level-1, namely $\beta = nln = \{H_0, H_2, H_3\}$ and pop an item from M for the next iteration. We continue to iterate based on the updated β and the value pop from M until reaching *Root*.

5.3 Update and compress Merkle tree

5.3.1 Update Merkle tree

After a block is committed, *MT* will be updated using the new state data to produce a new root, like state root in Ethereum [48]. The update process is also performed iteratively from the bottom to up. After getting trusted result, enclave uses the result and incoming multiproofs to update *MT* incrementally in parallel. Since the write set only involves the state data controlled by the same contract, and cross-contact transactions are not processed in the enclave, such updates are lightweight and friendly. Eventually, the updated tree structure will be synchronized to all nodes along with the trusted write sets via the secure channel of SGX. All state updates caused by cross-contract transactions will be completed after the *Follow* phase to ensure the consistency of nodes. Through incremental update based on SGX, the state update cost of all nodes is greatly reduced.

5.3.2 Compress Merkle tree

According to Algorithms 1 and 2, multi-value computation is used to generate proofs and verify data. Note that when the tree is processed to a specific height, the

input of each subsequent level is full, i.e., all necessary hashcodes are inputted from the lower level. If this height remains stable over multiple executions, it is better to compute a single hash for the level to obtain the root node, which can save a lot of computation cost. Furthermore, the root can be computed with a single hash from lower level if we can balance the cost of transmission and computation well. The transmission cost significantly varies under different access patterns. For example, in the worst case, where only one hashcode is inputted from the lower level, the transmission cost is the greatest with a single hash because all sibling node in the current level must be included in the proof. On the contrary, in the best case, where all hashcodes are inputted from the lower level, it is no need to provide any additional proofs and the transmission cost is minimized. To achieve this goal, we design a dynamic compression strategy to determine the height of a tree to be compressed based on recent data access pattern, aiming to further speed up the verification and update process. To reduce identification overhead, we use access pattern matching to quickly determine, which is relatively easy. Specifically, we pre-generate some judgment rules based on historical information, which record the number of necessary proofs for verification under different access patterns. Besides, the cost of computing and updating hashcode is also maintained for cost estimation during rule selections.

Example 5 For a *MT* consisting of four leaves, there is no need to provide additional proofs if all nodes are from the input set. Under this situation, the whole *MT* is computed from the leaf until root during verification and update, so we compute the hash value for $4 + 2 = 6$ times (four leaf nodes and two nodes in Level-1), and all hashcodes, for $4 + 2 + 1 = 7$ times (including the root). However, only four leaf nodes need to be calculated under a single hash to obtain the root. At the same time, only leaf nodes and the root node need to be considered when updating. In summary, using a single hash can greatly reduce the computation and update cost in this case.

Usually, the access pattern to a *MT* may be different during verification and update phases, so we need to consider these two phases separately. We mark the size of the compact multiproofs to be transferred for verification as P_V , the size of data to be computed for verification or update as C_V or C_U, and the number of hashcodes to be updated as W_U. We compute both costs as follows, where $T_{cost}(P_V)$ represents the cost to transfer proof of size P_V to the enclave, $C_{cost}(C_V + C_U)$ is the cost to compute the hashcode with the size $C_V + C_U$, and $W_{cost}(W_U)$ represents the cost to update W_U hashcodes.

$$Cost = T_{cost}(P_V) + C_{cost}(C_V + C_U) + W_{cost}(W_U)$$

The level can be compressed if and only if $\frac{Cost_{merkle}}{Cost_{compress}} \geq \alpha$. We can determine whether a level can be compressed or not by counting the recent access pattern. We will analyze a *MT* from top to bottom until it cannot be compressed, i.e., this level does not meet the compression condition. Finally, we obtain a compression rule $R = \{(Level_j, Level_i)\}$, i.e., for a *MT*, we compute the parent node in Level-i with a single hash from the Level-j.

The compression rule is initiated by *PrimaryNode* and attached to the block body to participate in consensus to ensure all nodes are under the same state tree structure. After consensus, all nodes begin to compress *MT* according to the rule contained in the block. Meanwhile, the compression rule is transmitted into the enclave to ensure correct validation. When applying modifications, we do not change the physical structure of the tree but replace original values with values recomputed under a compressed manner. Therefore, only a simple modification is needed for the previous algorithm. When obtaining proof, if the level hits a rule in R, we add all other values that are not inputted from the lower level to the proof. For verification and update, we compute the root with a single hash from the marked level.

Example 6 As shown in Fig. 10, the ordered index list γ of read set is $\{1, 4, 6\}$ and a compression rule is $R = \{(1, 3)\}$, which means we should compute the parent node at Level-3 using a single hash from Level-1. When generating multiproofs, we can handle Level-0 using Algorithm 1 and add H_1^1 for the proof at Level-1. During the validation phase, we can compute H_0^3 with a single hash of $\beta = \{H_0, H_2, H_3\}$ of Level-1 and H_1^1.

5.4 Concurrent Merkle tree

To increase the parallelism of generating multiproofs and verifying data integrity, we modify the structure of traditional *MT* and design a concurrent *MT* (*CMT*). For a dataset D, *CMT* consists of $k + 1$ subtrees, where each of the first k subtrees has 2^N (N is user-defined) leaf nodes, and the $(k + 1)$-th subtree has $m = |D| - k * 2^N$ leaves, where $m \in [0, 2^N)$. Compared with standard *MT*, each subtree in *CMT* is shorter, which can support a large dataset since proofs can be generated easily and data can be verified quickly by employing multi-threads. The first k subtrees are full *MT*s, and the last tree constructed by the remaining m data may be unfull. Take the roots of these $k + 1$ subtrees as a new dataset, we can continue to build more trees according to the above rules until the number of subtrees is less than 2^N. Meanwhile, we can get the last tree, i.e., root tree. In this way, for a large dataset, we can obtain multiple layers of *MT*. Figure 11 shows a simple example of *CMT* with $N = 2$ and $D = 7$. At the lowest layer (Layer-0), there are $k = 3$ full subtrees and $m = 1$ unfull subtree with one value H_6^0. Roots of the subtrees in Layer-0 are the leaves of the upper layer (Layer-1), so we continue to compute until reaching the root tree. For the known dataset, we set $k + 1$ equal to the number of cores to use the advantages of multi-core processor. After obtaining multiproofs, we only need to pay attention

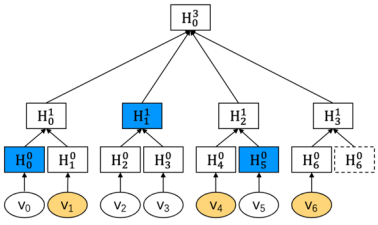


Fig. 10 The structure of a compressed *MT*

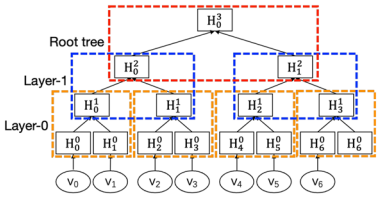


Fig. 11 An example of concurrent *MT*

to the involved subtrees. Once a tree is involved, its root must be calculated and become the input for the upper level. Through loop iterations, we will finally get all necessary proofs. Furthermore, for each subtree, we can independently calculate multiproofs and use them for verification based on Algorithms 1 and 2.

6 Analysis and optimization

6.1 Performance analysis

We analyze the performance of our framework in this section. The overall cost consists of two parts: computation cost and communication cost. The computation cost comes from the *Execution* phase and *Follow* phase. The communication cost is brought by the information sent from *ExecutionNode* to others.

For the traditional two-phase approach, we denote the pre-execution time of the first phase as C_p and the replay time of the second phase as C_r. For our work, each *ExecutionNode* executes a part of the block in the *Execution* phase. Assume the original block B is split into n micro-blocks $B_1, B_2, \ldots, B_n$, and the rest data block is B_r, which contains all unexecuted transactions including cross-contract transactions. $ExecutionNode_i$ deals with $\frac{|B_i|}{|B|}$ of the whole data. Let $M(B)$ denote the cost of data transfer for a block B between the enclave and the the untrusted memory, and C_c the execution cost of cross-contract transactions. Therefore, the total cost of *Execution* phase for $ExecutionNode_i$ is $\frac{|B_i|}{|B|}(C_p + C_r) + M(B_i)$. In the *Follow* phase, a node decrypts the result from others and executes all rest cross-contract transactions. Let $D(B_i)$ denote the cost of decrypting a data block B_i, then the cost of the *Follow* phase is $D(B - B_r) + C_c(B_r)$. Therefore, the computation cost is expressed as follows.

$$\max_{1 \le i \le n} \left(\frac{|B_i|}{|B|}(C_p + C_r) + M(B_i) \right) + D(B - B_r) + C_c(B_r)$$

From the above equation, we find: (i) When the contracts in the block are well balanced, as the number of nodes increases, the execution can scale well among nodes; (ii) When all transactions in a block are cross-contract transactions, the total cost is equal to the cost of executing all transactions on each node. The equation also suggests that the performance improvement should focus on the distribution of transactions in a block and data exchange efficiency between the untrusted memory and enclave.

For the communication cost, all results need to propagate by Gossip protocol after the *Execution* phase, and the communication cost is $T(B - B_r)$. When all contracts are cross-contract, the additional communication cost is 0. Therefore, combining the cost of the computation and communication, the cost of our solution is equal to that of the traditional serial execution solution when sharding is not possible. Otherwise, our solution can improve performance by high inter-node concurrency and intra-node concurrency.

6.2 Security analysis

6.2.1 Fake data

There are at most f malicious out of n nodes, just like the assumption of PBFT [52] where $n \geq 3f + 1$. Each malicious node may send, drop, modify, or record arbitrary messages at any time during the contract deployment and invocation. Thanks to the confidentiality guarantee provided by SGX, all possible malicious behaviors during the pre-execution can be detected during the validation in the enclave. Therefore, the results obtained by a *FollowNode* from others via the secure channel of SGX must be correct, i.e., not maliciously tampered with. As stated above, all data necessary for validation in the enclave are from untrusted memory. Under this case, a malicious node may tamper with local data, leading to the incorrect read set being transferred into the enclave. Then, an incorrect result can be generated based on this incoming wrong read set in the enclave, resulting in wrong results being endorsed successfully and synchronized to other nodes. To avoid this, we first generate a *MT* using all initial state data, whose root is signed when consensus is reached. Then $2f + 1$ signatures on this root are collected as the proof. This signature collection process can be tied to the consensus of blocks and does not impact the overall performance. SGX uses these $2f + 1$ signatures to verify whether the root is correct or not and then uses the verified root and corresponding multiproofs to verify the correctness of incoming read set. This verification method is correct because malicious nodes cannot forge $2f + 1$ correct root signatures.

6.2.2 Replay attack

Due to the existence of Byzantine nodes, the initial read set can be replaced with an old version when getting the compact validation information in untrusted memory, making stale data read in trusted memory. To avoid replay attack, each message to be transferred to the enclave will be endowed with a monotonic counter. A trusted monotonic counter also is maintained in the enclave to protect the latest version of the round. To detect and defend against the replay attack, we use SGX monotonic counter service to guarantee the freshness of incoming data. Comparing the values of these two counters can ensure the incoming data is new with the same version, and transactions will not be executed if the data is old.

6.3 Combine with PBFT

Currently, permissioned blockchain systems usually use a BFT-like consensus algorithm, which includes three rounds of communication and one-third of the fault-tolerant (i.e., the number of malicious nodes is less than one-third of the total number of the system). Thanks to these characteristics, our execution framework can work fine with BFT-like protocols. We further integrate our execution framework with PBFT and analyze its correctness theoretically. Figure 12 shows the whole process flow, where f is the maximal number of malicious nodes the system can tolerate.

Firstly, *PrimaryNode* uses the strategy proposed in Sect. 4.1 to package a batch of transactions sent by clients into a block. Then, it broadcasts a pre-prepared message to all *ReplicaNode*s along with the pre-allocated information. Once a node is prepared, i.e., receiving $2f$ matching prepare messages (excluding itself), it starts *Execution* phase. The node firstly executes the contract block assigned to it using concurrent execution policy, and obtains the corresponding validation information in the untrusted memory. Then, this validation information is passed to the enclave to concurrent validate using our proposed scheme. All cross-contract transactions are identified and aborted during *Execution* phase, which will be attached to the encrypted results and sent to all replicas via the secure channel of SGX. After validation, a commit message will be sent to other nodes. When a replica receives $2f$ matching commit message (excluding itself), it starts *Follow* phase. During the *Follow* phase, all nodes firstly update the local state by decrypting the corresponding state sets and then use deterministic concurrency protocol to execute all unexecuted transactions it collects.

The process of blocks packaged and tasks pre-allocated by *PrimaryNode* does not affect the correctness of PBFT, because it occurs before the consensus process. Meanwhile, the embedded SGX-based two-phase execution also does not affect the correctness. If the node is malicious, the validation process in the trusted memory will not pass, and any incorrect result will not be confirmed. Any malicious action can be discovered during the validation in the enclave. After a timeout, PBFT triggers the view change protocol.

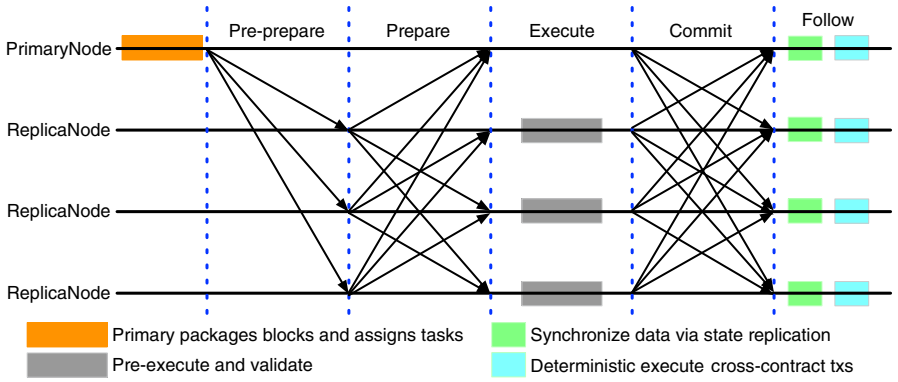


Fig. 12 Workflow when integrating with PBFT

7 Experiments

We implement a prototype system that integrates all techniques proposed above with an open-source PBFT framework BFT-SMaRt [22] and evaluate its performance in this section.

7.1 Experiment setup

All experiments were conducted upon a cluster of 4 virtual machines based on UCloud-N2, and the configuration of each one is shown in Table 2. Besides, the project is built in the debug simulation mode of Intel SGX withe 128MB of EPC. All codes in untrusted memory and the enclave are written in Java and C++ respectively, and communicate via JNI.

We use SmallBank as the workload for testing, which is widely used to evaluate blockchain systems [1, 8, 39]. To simulate the real workload properly, we extend the original SmallBank benchmark by adding two new transactions: *SendPayment* and *Query*, where *SendPayment* transfers money from one account to another one as shown in Fig. 1, and *Query* reads the checking and the saving account of a user. The database is initiated with 1000 k customers, including 1000k checking and 1000 k saving accounts. Before the start of each round of experiments, the system is warmed up for three minutes, and all experimental figures show an average of 10 runs. At the same time, we fix the size of a subtree in *CMT* as 2^{10} (the number of leaf nodes involved). During each run, we trigger these six transactions in a random manner, where a certain probability P_w is designed to decide whether pick one of five writing transactions or not, and the reading transaction is picked with probability $1 - P_w$. We set the default setting of P_w to $p = \frac{1}{6}$, which means the workload will be generated on these six transactions evenly, e.g., when the workload size is X, the amount allocated to each of these six transactions is $X \cdot p = \frac{X}{6}$. Data access pattern follows a Zipfian distribution to simulate the data skew situation. Note that the greater the skew value, the higher the transaction conflict rate. Note that the throughput in the experiments below includes all transactions of a block, and the default setting of the failure rate is 0.

In the following experiments, we use *SE*, *CE*, *CEIS*, and *CEWS* to denote serial execution, concurrent execution, concurrent execution in SGX (put execution completely in SGX), and our concurrent execution with SGX respectively. Due to the size limitation, it is infeasible to put all data into the enclave. Given this, if we put the execution of smart contracts in SGX, it will inevitably introduce a large number of enclave transitions and encryption and decryption, which is costly to the system. Therefore, to further reduce the overhead introduced when executing in SGX, we use two-phase locking

Table 2 Machine configuration

Resource type	CPU core	Memory	System	JDK	Ethernet	SGX SDK [53]
Quota	16	16 GB	Ubuntu 18.04	v11	1 GB	v2.13

(2PL) protocol to implement *CEIS* to avoid additional overhead of operations such as validation and rollback in *OCC*.

7.2 Concurrency performance

We evaluate the concurrency and scalability of the system, including intra-node concurrency under multiple threads and inter-node concurrency under different number of smart contracts.

7.2.1 Varying the number of threads

Figure 13 shows throughput and latency of the system under different numbers of threads upon 4 smart contracts, 2048 transactions per block, and $P_w = p$ under uniform distribution when the number of threads varies from 1 to 16. Obviously, the throughput of *CE*, *CEIS*, and *CEWS* rises with the increase of thread number. The latency decreases as the thread number grows and is smaller than *SE*. Experimental results show the performance of *CEWS* under multi-threading is significantly better than other methods. However, the system performance will reach the upper limit when the number of threads reaches a threshold. As shown in the Fig. 13, the throughput and latency of *CEWS* and *CE* towards stability when the number of threads reaches 8. However, when the number of threads is greater than 12, the system throughput starts to fluctuate significantly. The reason is that the competition of background threads has already affected the system's current state. Therefore, we fixed the number of threads in the remaining experiments to 12 to obtain high system throughput.

7.2.2 Varying the number of smart contracts

We evaluate the scalability of *CEWS* by varying the number of smart contracts from 1 to 4 because four machines are used in our experiment, the number of transactions per block is 2048, and $P_w = p$ under uniform distribution with fixed 12 threads. Figure 14 reports throughput and latency with the number of smart contracts. Under all situations, *SE* and *CE* are almost unchanged in throughput and latency because they

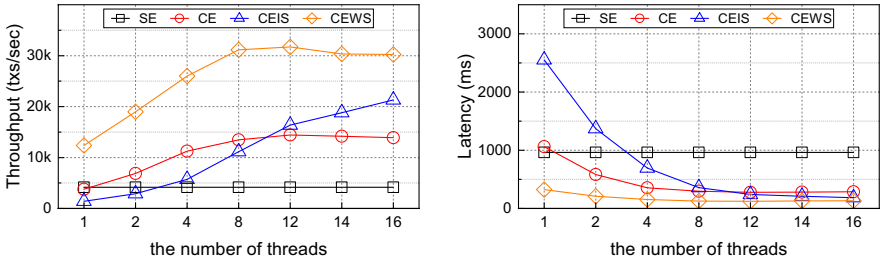
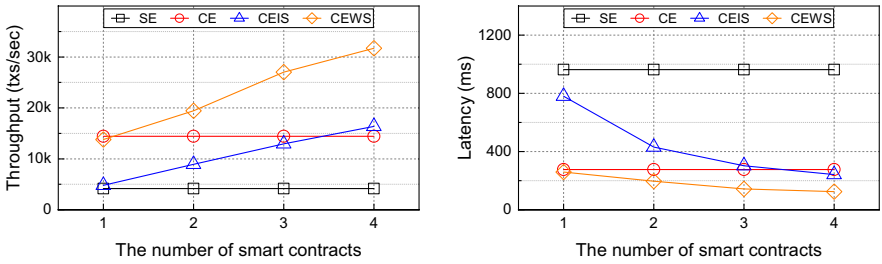


Fig. 13 Performance against the number of threads



(a) Throughput **(b)** Latency

Fig. 14 Performance against the number of smart contracts

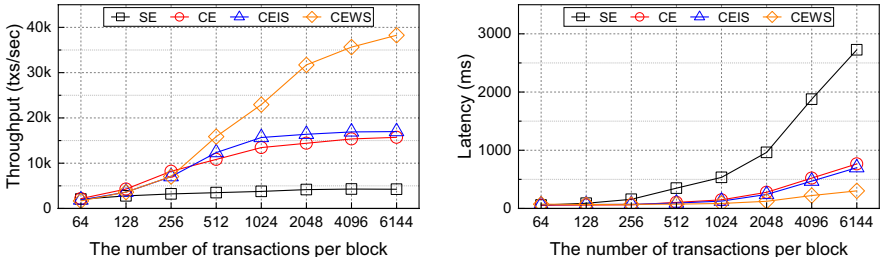
do not consider the division of transactions. According to Fig. 14a, the throughput of *CEIS* and *CEWS* continues to rise and does not show any flattening trend. Compared with *CEIS*, *CEWS* performs better because of less enclave interaction and transmission during execution. Therefore, as the number of contracts increases, *CEWS* can complete the calculation with higher degree of parallelism. At the same time, the latency of *CEWS* is smaller than others, as shown in Fig.14b. For the remaining experiments, we fix the number of smart contracts to 4 to achieve high overall throughput.

7.3 Workload distribution

We evaluate the performance of our system under different workloads, including block size, workload skew, and the number of cross-contract transactions.

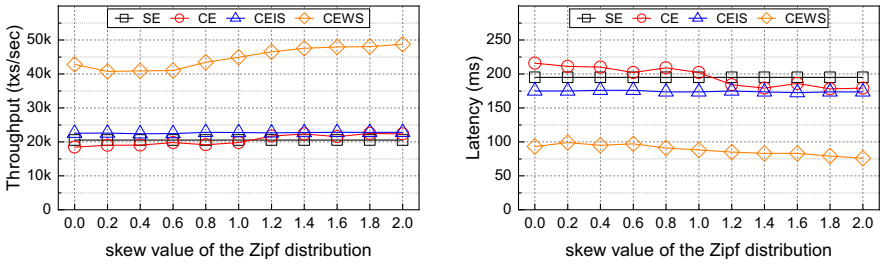
7.3.1 Varying the number of transactions per block

Figure 15 reports the throughput and latency upon 4 smart contracts, $P_w = p$, and 16 threads over uniform distribution when the number of transactions per block varies from 64 to 6144. Both throughput and latency of *SE*, *CE*, *CEIS*, and *CEWS* increase as the block size increases, because enhancing block size lowers



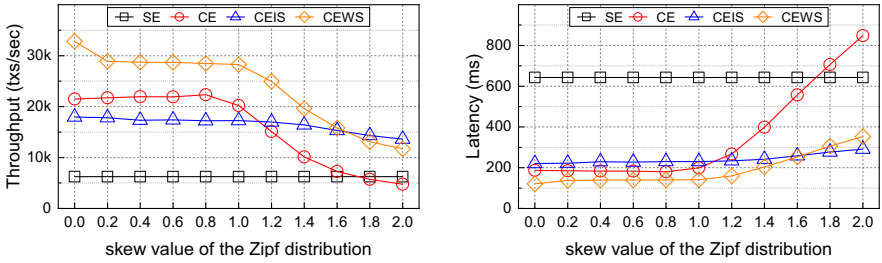
(a) Throughput **(b)** Latency

Fig. 15 Performance against the number of transactions



(a) Throughput **(b)** Latency

Fig. 16 Performance against the value of skew ($P_w = 5\%$)



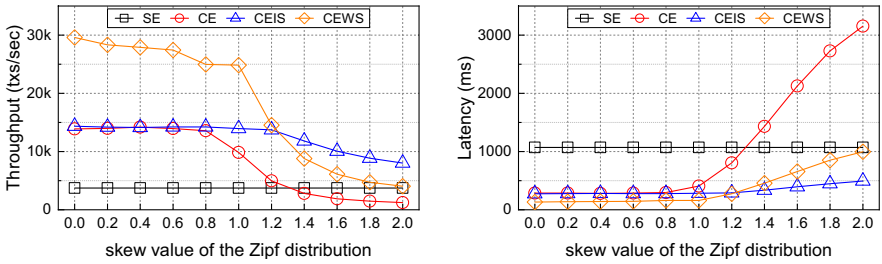
(a) Throughput **(b)** Latency

Fig. 17 Performance against the value of skew ($P_w = 50\%$)

network communication, while raises execution overhead. Obviously, as the block size increases, *CEWS* gains higher throughput and lower latency than others. The throughput of *CEWS* does not have any tendency to flatten here, but *CE* and *CEIS* start to flatten when the block size is greater than 2048. This clearly shows the effectiveness of our proposed method, which benefits from larger blocks. For the compromise principle, we use a block size of 2048 transactions for the rest experiments because the throughput and latency of *SE*, *CE*, *CEIS* and *CEWS* are within an acceptable range under this setting.

7.3.2 Varying the value of skew

We vary the probability of picking a writing transaction over the reading transaction in three different parameters $P_w = 5\%$ (read-heavy), $P_w = 50\%$ (balanced) and $P_w = 95\%$ (write-heavy), to evaluate the throughput of *CEWS*. This upon fixed 4 smart contracts, 2048 transactions per block and 16 threads under varying skew from 0.0 to 2.0. Figures 16, 17 and 18 shows the results under different P_w. When $P_w = 5\%$, *CEWS* has higher throughput and lower latency than others. At the same time, under this read-heavy workload, concurrent execution does not bring significant performance improvement, so *SE*, *CE* and *CEIS* have similar effect. Moreover, under $P_w = 50\%$ and $P_w = 95\%$, after the skew parameter is smaller than 1.0, the throughput of *CE* and *CEWS* is relatively high because of smaller potential conflict



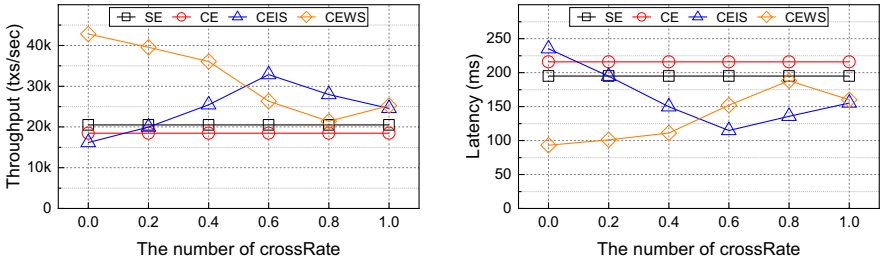
(a) Throughput **(b)** Latency

Fig. 18 Performance against the value of skew ($P_w = 95\%$)

between transactions. However, for higher skew (skew value is greater than 1.0), the throughput of *SE* and *CE* declines sharply because of higher abort rate with the *OCC* protocol. On the contrary, the skew value basically has no effect on *CEIS*, because it uses 2PL instead of *OCC* protocol, which is lower sensitive to high skew workloads.

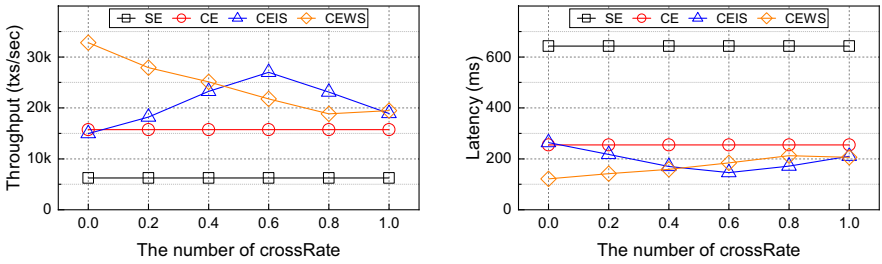
7.3.3 Varying the number of cross-contract transactions

We tested the system performance for different proportion of cross-contract transactions in the block. This fixed 4 smart contracts, 2,048 transactions per block, $P_w = p$ and 16 threads, and varied the percentage of cross-contract transactions in the batch from 0.0 to 1.0. Figures 19, 20 and 21 shows the results under different P_w. As *crossRate* grows, *SE* and *CE* remain essentially unchanged because they both need all nodes to execute all transactions in a block. However, in *CEWS* and *CEWS*, a batch of transactions is sliced and assigned to several different nodes for execution, and the increase of *crossRate* inevitably affects system performance because cross-contract transactions involve multiple nodes and cannot be simply executed by a node. Even so, with the growth of *crossRate*, *CEIS* and *CEWS* still perform better than *CE* because we use a deterministic concurrency approach to process all identified unexecuted transactions. The throughput of *CEWS* first drops and then rises because the concurrency among nodes has a major impact on the system at the beginning. However, when cross-contract transactions increase,



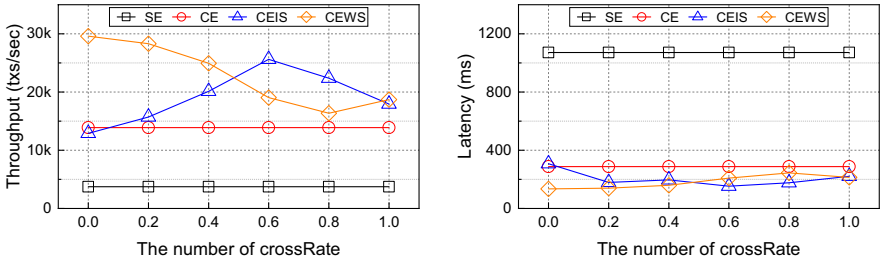
(a) Throughput **(b)** Latency

Fig. 19 Performance against the number of cross-contract transactions ($P_w = 5\%$)



(a) Throughput **(b)** Latency

Fig. 20 Performance against the number of cross-contract transactions ($P_w = 50\%$)



(a) Throughput **(b)** Latency

Fig. 21 Performance against the number of cross-contract transactions ($P_w = 95\%$)

this advantage is weakened, and the throughput naturally decreases. When it reaches a certain level, the deterministic concurrency of cross-contract transactions has a major impact on the system, so the throughput increases. Compared with *CEWS*, the throughput of *CEIS* first increases and then decreases because there are a large number of enclave transitions and high data encryption and decryption costs during the execution of *CEIS*. When cross-contract transactions increase, these costs will decrease. However, when *crossRate* reaches a certain threshold, since most transactions are cross-contracts, the concurrency among nodes decreases, and the cost of deterministic concurrency increases, which leads to a decrease in system performance. When across-contract transactions reach about 50%, fewer transactions are executed in the *Execution Phase*, leading the extra cost being more significant than execution cost.

8 Related work

We review the latest researches close to ours in this section, including smart contract concurrent execution, combining smart contracts with SGX, and Merkle multiproofs.

8.1 Concurrent smart contract execution

Due to the existence of Byzantine nodes, the execution of a smart contract at each node must produce exactly the same final state. Given this, none of the existing mature blockchain systems allow any concurrency in transaction execution, i.e., nodes execute smart contract transactions serially in the order that appears in a block. Serial execution is the simplest way to ensure the consistency of execution results among nodes, but it does not take advantage of modern multi-core processors, which in turn slows down the overall performance of the system. Given above, smart contracts concurrent execution has been studied recently to replace the serial execution method. Dickerson et al. propose the concurrent execution method of smart contracts, where the primary uses a two-phase lock protocol to ensure serializability in the first phase, and a concurrent schedule is recorded during execution and uploaded to replicas for validating in the second phase [11]. Anjana et al. replace the pessimistic lock protocol with an optimistic concurrency control protocol at the primary, independent and parallel validation is used to improve efficiency in the second validation phase [10]. Zhang et al. allow the primary to use any concurrency control protocol to get a conflict-serializable schedule without conflict assumption, and a read-write set is transmitted to replicas for re-executing in the second phase [12]. Jin et al. use batching *OCC* based on conflict graph segmentation to optimize both the execution efficiency of primary and the transaction replay efficiency of replicas [8]. However, the aforementioned two-phase concurrent execution approaches can only achieve concurrent execution of intra-node situations, not inter-node situations, because the second phase begins after the first phase completes.

We focus on all the above problems and propose the SGX-based two-phase execution framework to achieve high-performance concurrent execution for smart contracts, as summarized in Table 3. Among these, our work achieves both intra- and inter-node concurrency for smart contracts and supports any concurrent execution method adopted before. The consistency among nodes can be achieved via simple state replication rather than re-execution.

Table 3 Concurrent execution framework of smart contracts

	Phase 1	Phase 2	Intra-node concurrency	Inter-nodes concurrency
Dickerson's [11]	Pessimistic	Re-executing	✓	×
Zhang's [12]	Any	Re-executing	✓	×
Anjana's [10]	Optimistic	Re-executing	✓	×
Jin's [8]	Optimistic	Re-executing	✓	×
Our work	Any	State replication	✓	✓

8.2 Smart contracts with Intel SGX

As SGX has been widely supported in Intel Core processor platforms of the sixth generation and higher, several works try to address the security and privacy issues in smart contract design by using it. Hawk [54] provides a confidential execution environment for contracts by using cryptography and SGX. Town Crier [55] provides reliable data from trusted network servers to smart contracts through trusted hardware such as SGX. Shadoweth [56] places the execution and storage of private contracts in the trusted region off-chain and only puts the process of verification on-chain. Microsoft introduces an open-source blockchain framework called Coco [57, 58], where each node is protected by SGX, and validated before joining the network. However, due to the limited memory of SGX, it is infeasible and impossible for Microsoft to pre-load all key components of the blockchain system (PBFT consensus, in-memory database on the chain, and RPC services for interaction) into SGX. Ekiden [59] tries to make all smart contracts execute in SGX, but it only suits for the case with a limited number of transactions, since more transactions may exceed the limited trusted memory. In a word, none of the above work considers how to hasten smart contract execution in SGX, not to mention the performance limitations of SGX. Note that our work takes the interest of both and is the first piece of work about smart contract concurrent execution in SGX.

8.3 Merkle multiproofs

Merkle tree (*MT*) is widely used to verify the integrity of data in blockchain [47, 48]. Although *MT* and its variants have been studied for a long time, most of them only address single-value validation, i.e., authenticate all challenged leaf nodes one by one. Recently, some studies turn to the problem of sparse Merkle multiproofs, in which the proofs of all values are generated and verified together [51]. To further reduce communication cost, [49] compresses the obtained sparse multiproofs further, which only records the index of necessary elements other than that for every non-leaf hash values. To improve the efficiency of integrity verification, [50] designs a new authenticated *MT* to batch-verify multiple values together. Specifically, an auxiliary authentication table ($AAT_{k \cdot m}$) is used to maintain all needed auxiliary authentication information for all k verified leaf on a Merkle tree with height m. Thus, multiple values can be verified together based on $AAT_{k \cdot m}$. However, it needs to calculate $k \cdot m$ times to build *AAT* and must maintain *NULL* for all unneeded items. Thus, the cost of generating and maintaining an *AAT* for each verification is high, especially when the *MT* and the verified set are large. In a word, none above can generate and verify sparse multiproofs for a large *MT* in parallel.

9 Conclusion

In this paper, we present an efficient SGX-based execution framework of smart contracts for permissioned blockchain aiming at higher parallelism and throughput. We focus on the performance problem of concurrent execution of smart contracts on

SGX-equipped nodes due to the limited size of the enclave, and the expensive cost of enclave transition and enclave paging. We devise a pre-execution mechanism for smart contracts in untrusted memory to batch fetch all state data that a smart contract needs to access to minimize the expensive overhead of enclave transitions during smart contract execution. And a mixed *OCC* protocol is proposed to accelerate execution. Meanwhile, we propose a novel mechanism based on *MT* to efficiently generate compact Merkle multiproofs and verify the data in parallel, even with a large dataset.

Acknowledgements This work is partially supported by the National Science Foundation of China (U1811264, U1911203, and 61972152) and Guangxi Key Laboratory of Trusted Software (kx202005). The authors would like to thank the anonymous reviewers for their valuable feedback.

References

1. Dinh, T.T.A., Wang, J., Chen, G., Liu, R., Ooi, B.C., Tan, K.L.: Blockbench: a framework for analyzing private blockchains. In: Proceedings of the 2017 ACM International Conference on Management of Data, pp. 1085–1100. ACM (2017)
2. Gupta, S., Hellings, J., Sadoghi, M.: Fault-tolerant distributed transactions on blockchain. Synth. Lect. Data Manag. **16**(1), 1–268 (2021)
3. Zhu, Y., Zhang, Z., Jin, C., Zhou, A., Yan, Y.: SEBDB: semantics empowered blockchain database. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE), pp. 1820–1831. IEEE (2019)
4. Gupta, S., Hellings, J., Sadoghi, M.: RCC: resilient concurrent consensus for high-throughput secure transaction processing. In: 2021 IEEE 37th International Conference on Data Engineering (ICDE), pp. 1392–1403. IEEE (2021)
5. Hellings, J., Sadoghi, M.: Byshard: sharding in a Byzantine environment. Proc. VLDB Endow. **14**(11), 2230–2243 (2021)
6. Miller, A., Xia, Y., Croman, K., Shi, E., Song, D.: The honey badger of BFT protocols. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 31–42. ACM (2016)
7. Stathakopoulou, C., David, T., Pavlovic, M., Vukolić, M.: Mir-BFT: High-throughput robust BFT for decentralized networks. arXiv preprint (2019). http://arxiv.org/abs/1906.05552
8. Jin, C., Pang, S., Qi, X., Zhang, Z., Zhou, A.: A high performance concurrency protocol for smart contracts of permissioned blockchain. IEEE Trans. Knowl. Data Eng. (2021). https://doi.org/10.1109/TKDE.2021.3059959
9. Vukolić, M.: Rethinking permissioned blockchains. In: Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts, pp. 3–7. ACM (2017)
10. Anjana, P.S., Kumari, S., Peri, S., Rathor, S., Somani, A.: An efficient framework for optimistic concurrent execution of smart contracts. In: 2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 83–92. IEEE (2019)
11. Dickerson, T., Gazzillo, P., Herlihy, M., Koskinen, E.: Adding concurrency to smart contracts. In: Proceedings of the ACM Symposium on Principles of Distributed Computing, pp. 303–312. ACM (2017)
12. Zhang, A., Zhang, K.: Enabling concurrency on smart contracts using multiversion ordering. In: Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint International Conference on Web and Big Data, pp. 425–439. Springer (2018)
13. Costan, V., Devadas, S.: Intel SGX explained. IACR Cryptol. ePrint Arch. **2016**(086), 1–118 (2016)
14. Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'keeffe, D., Stillwell, M.L., et al.: {SCONE}: Secure Linux containers with intel {SGX}. In: 12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16), pp. 689–703 (2016)

15. Tsai, C.C., Porter, D.E., Vij, M.: Graphene-SGX: a practical library {OS} for unmodified applications on {SGX}. In: 2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17), pp. 645–658 (2017)
16. Gjerdrum, A.T., Pettersen, R., Johansen, H.D., Johansen, D.: Performance principles for trusted computing with intel SGX. In: International Conference on Cloud Computing and Services Science, pp. 1–18. Springer (2017)
17. Weichbrodt, N., Aublin, P.L., Kapitza, R.: sgx-perf: a performance analysis tool for Intel SGX enclaves. In: Proceedings of the 19th International Middleware Conference, pp. 201–213. ACM (2018)
18. Performance Considerations for Intel SGX Applications. https://software.intel.com/content/dam/develop/public/us/en/documents/intel-sgx-performance-considerations.pdf
19. Weisse, O., Bertacco, V., Austin, T.: Regaining lost cycles with hotcalls: A fast interface for SGX secure enclaves. ACM SIGARCH Comput. Archit. News **45**(2), 81–93 (2017)
20. Merkle, R.C.: Method of providing digital signatures. US Patent 4,309,569 (1982)
21. Merkle, R.C.: A certified digital signature. In: Conference on the Theory and Application of Cryptology, pp. 218–238. Springer (1989)
22. BFT-SMaRt. https://github.com/bft-smart/library
23. Fang, M., Zhang, Z., Jin, C., Zhou, A.: High-performance smart contracts concurrent execution for permissioned blockchain using SGX. In: 2021 IEEE 37th International Conference on Data Engineering (ICDE), pp. 1907–1912. IEEE (2021)
24. Szabo, N.: Formalizing and securing relationships on public networks. First Monday (1997). https://doi.org/10.5210/fm.v2i9.548
25. Wood, G., et al.: Ethereum: a secure decentralised generalised transaction ledger. Ethereum Proj. Yellow Pap. **151**(2014), 1–32 (2014)
26. Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., et al.: Hyperledger fabric: a distributed operating system for permissioned blockchains. In: Proceedings of the Thirteenth EuroSys Conference, pp. 1–15 (2018)
27. Solidity. https://solidity.readthedocs.io/en/latest/
28. Minkin, M., Silberstein, M.: Improving performance and security of Intel SGX. Ph.D. thesis, Computer Science Department, Technion (2019)
29. Amiri, M.J., Agrawal, D., El Abbadi, A.: Sharper: sharding permissioned blockchains over network clusters. In: Proceedings of the 2021 International Conference on Management of Data, pp. 76–88 (2021)
30. Wüst, K., Matetic, S., Egli, S., Kostiainen, K., Capkun, S.: ACE: asynchronous and concurrent execution of complex smart contracts. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pp. 587–600 (2020)
31. Yang, Y., Papadias, D., Papadopoulos, S., Kalnis, P.: Authenticated join processing in outsourced databases. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, pp. 5–18 (2009)
32. Nilsson, A., Bideh, P.N., Brorsson, J.: A survey of published attacks on intel SGX. arXiv preprint. (2020). http://arxiv.org/abs/2006.13598
33. van Schaik, S., Minkin, M., Kwong, A., Genkin, D., Yarom, Y.: Cacheout: Leaking data on intel CPUs via cache evictions. In: 2021 IEEE Symposium on Security and Privacy (SP), pp. 339–354. IEEE (2021)
34. Wang, W., Chen, G., Pan, X., Zhang, Y., Wang, X., Bindschaedler, V., Tang, H., Gunter, C.A.: Leaky cauldron on the dark land: understanding memory side-channel hazards in SGX. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 2421–2434 (2017)
35. Al-Bassam, M., Sonnino, A., Bano, S., Hrycyszyn, D., Danezis, G.: Chainspace: a sharded smart contracts platform. In: 25th Annual Network and Distributed System Security Symposium (NDSS) (2018)
36. Chainspace. https://github.com/xuperchain/xuperchain
37. Gupta, S., Rahnama, S., Sadoghi, M.: Permissioned blockchain through the looking glass: architectural and implementation lessons learned. In: 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS), pp. 754–764. IEEE (2020)
38. Cahill, M.J., Röhm, U., Fekete, A.D.: Serializable isolation for snapshot databases. ACM Trans. Database Syst. **34**(4), 1–42 (2009)

39. Sharma, A., Schuhknecht, F.M., Agrawal, D., Dittrich, J.: Blurring the lines between blockchains and database systems: the case of hyperledger fabric. In: Proceedings of the 2019 International Conference on Management of Data, pp. 105–122. ACM (2019)
40. Chacko, J.A., Mayer, R., Jacobsen, H.A.: Why do my blockchain transactions fail? A study of hyperledger fabric. In: Proceedings of the 2021 International Conference on Management of Data, pp. 221–234 (2021)
41. Kim, J.Y., Lee, J., Koo, Y., Park, S., Moon, S.M.: Ethanos: efficient bootstrapping for full nodes on account-based blockchain. In: Proceedings of the Sixteenth European Conference on Computer Systems, pp. 99–113 (2021)
42. Ponnapalli, S., Shah, A., Banerjee, S., Malkhi, D., Tai, A., Chidambaram, V., Wei, M.: RainBlock: faster transaction processing in public blockchains. In: 2021 USENIX Annual Technical Conference (USENIX ATC 21), pp. 333–347 (2021)
43. Guide, P.: Intel® 64 and IA-32 architectures software developer's manual. Volume 3B: System programming Guide, Part **2**(11) (2011)
44. Orenbach, M., Lifshits, P., Minkin, M., Silberstein, M.: Eleos: Exitless OS services for SGX enclaves. In: Proceedings of the Twelfth European Conference on Computer Systems, pp. 238–253 (2017)
45. Slither. https://github.com/crytic/slither
46. Prasaad, G., Cheung, A., Suciu, D.: Handling highly contended OLTP workloads using fast dynamic partitioning. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pp. 527–542 (2020)
47. Bitcoin. https://github.com/bitcoin/bitcoin
48. Ethereum. https://github.com/ethereum
49. Ramabaja, L., Avdullahu, A.: Compact Merkle multiproofs. arXiv preprint (2020). arXiv:2002.07648
50. Rao, L., Zhang, H., Tu, T.: Dynamic outsourced auditing services for cloud storage based on batch-leaves-authenticated Merkle hash tree. IEEE Trans. Serv. Comput. **13**(3), 451–463 (2017)
51. Sparse Merkle Multiproofs. https://medium.com/@jgm.orinoco/understanding-sparse-merkle-multiproofs-9b9f049e8f08
52. Castro, M., Liskov, B., et al.: Practical byzantine fault tolerance. OsDI **99**, 173–186 (1999)
53. Intel SGX SDK. https://software.intel.com/en-us/sgx/sdk
54. Kosba, A., Miller, A., Shi, E., Wen, Z., Papamanthou, C.: Hawk: the blockchain model of cryptography and privacy-preserving smart contracts. In: 2016 IEEE Symposium on Security and Privacy (SP), pp. 839–858. IEEE (2016)
55. Zhang, F., Cecchetti, E., Croman, K., Juels, A., Shi, E.: Town Crier: an authenticated data feed for smart contracts. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 270–282. ACM (2016)
56. Yuan, R., Xia, Y.B., Chen, H.B., Zang, B.Y., Xie, J.: Shadoweth: private smart contract on public blockchain. J. Comput. Sci. Technol. **33**(3), 542–556 (2018)
57. Coco-Framework. https://github.com/Microsoft/CCF
58. Russinovich, M., Ashton, E., Avanessians, C., Castro, M., Chamayou, A., Clebsch, S., Costa, M., Fournet, C., Kerner, M., Krishna, S., et al.: CCF: a framework for building confidential verifiable replicated services. Technical Report MSR-TR-2019-16, Microsoft (2019)
59. Cheng, R., Zhang, F., Kos, J., He, W., Hynes, N., Johnson, N., Juels, A., Miller, A., Song, D.: Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts. In: 2019 IEEE European Symposium on Security and Privacy (EuroS &P), pp. 185–200. IEEE (2019)