Jenga: Orchestrating Smart Contracts in Sharding-Based Blockchain for Efficient Processing

Mingzhe Li[†*], You Lin[*], Jin Zhang[*✉], Wei Wang[†✉]
[†]*Computer Science and Engineering Department, Hong Kong University of Science and Technology*
[*]*Guangdong Provincial Key Laboratory of Brain-inspired Intelligent Computation,*
Research Institute of Trustworthy Autonomous Systems,
Computer Science and Engineering Department, Southern University of Science and Technology
mlibn@cse.ust.hk, liny2021@mail.sustech.edu.cn, zhangj4@sustech.edu.cn, weiwa@cse.ust.hk

***Abstract*—Sharding is a promising way to achieve blockchain scalability, increasing the throughput by partitioning nodes into multiple smaller groups, splitting the workload. However, when tackling the increasingly important smart contracts, existing blockchain sharding protocols do not scale well. They usually require complex multi-round cross-shard consensus protocols for contract execution and extensive cross-shard communication during state transmission, mainly because that each shard stores and executes an isolated, disjoint subset of contracts. In this paper, we present Jenga, a novel sharding-based approach for efficient smart contract processing. Its main idea is to break the isolation between shards by orchestrating the logic storage, state storage, and execution of smart contracts. In Jenga, *all shards share the logic for all contracts*. Therefore, multiple contracts involved in a smart contract transaction can be executed together by the same shard within one round. Moreover, *different shards store distinct states* (named state shards), several "orthogonal" execution channels are established based on the state shards, where each channel overlaps with all shards. Each node simultaneously belongs to a shard and an "orthogonal" channel, *different channels execute different contracts*. Therefore, via the overlapped nodes, the contract states can be directly broadcast between the state shards and the execution channels without additional cross-shard communication. We implement Jenga and evaluation results show that it provides outstanding performance gains in terms of throughput and transaction confirmation latency.**

I. Introduction

Blockchain has been instrumental for enabling decentralized digital currencies, such as Bitcoin [20]. With the wide-spread adoption of *smart contracts*, the applications of blockchain have been expanded to broader scenarios (e.g., DeFi [1], Metaverse [10], etc.). Smart contracts are self-enforcing, self-executing programs governing an interaction between mutually distrusting parties [22]. A smart contract can be deployed in blockchain, where all blockchain nodes (aka miners) store its *state (aka data)* and execution *logic (aka function)* [12]. Smart contracts stand in an *increasingly important* position in blockchain systems. More and more transactions invoke smart contracts (named smart contract transaction) to handle complex logic. It is observed that smart contract transactions take around 70% of all the recent Ethereum [28] transactions, and this proportion is expected to grow in the future [22].

Scalability is an urgent concern in blockchain [26], and *sharding* is a promising technique that can significantly improve the scalability of blockchain [17]. In legacy blockchain systems, each node stores the whole blockchain state and processes all transactions, and every consensus message needs to be broadcast in the whole network [13]. In contrast, sharding divides nodes into multiple groups called shards, each storing a distinct subset of the whole blockchain state, executing distinct transactions and reaching consensus in parallel [29]. Via such parallelization, sharding increases the scalability of legacy blockchain in terms of storage, computation, network, and transactions per second (TPS) [29].

However, most existing sharding works [14], [15], [17], [27], [29], [30] focus exclusively on handling the simplest kind of transactions—account-to-account transfer of funds, while *ignoring how to efficiently handle smart contract transactions.* There are considerable differences between processing smart contract transactions and processing regular fund transfer transactions. Typically, a smart contract has complex execution logic, and executing it requires multiple states from accounts and contracts. Furthermore, a smart contract transaction usually invokes multiple different contracts. As a result, compared to the regular transfer transaction which contains two account states and one step, a smart contract transaction usually requires *multiple execution steps* between different contracts to execute the whole *complex logic* it has invoked. Additionally, the processing of a smart contract transaction involves *more states*.

Only a few existing proposals focus on handling smart contracts in blockchain sharding, and their solutions impose various limitations. For instance, some works [4], [9], [25] require all smart contracts to be processed in one specific shard, resulting in poor ability to process transactions in parallel. To improve the scalability, some solutions allow smart contracts to be deployed and processed in different shards. However, among them, some can only tackle smart contract transactions involving single contract and one step execution [2], [7], [22]. Other solutions support processing transactions with multiple contracts [11], [23]. Nevertheless, they process contract transactions with complex, multi-round cross-shard consensus protocols, causing performance degradation in terms of throughput and latency.

Why existing blockchain sharding systems suffer from low throughput and high latency when handling smart contracts? We find that one potential reason is the *isolation between shards.* Particularly, different smart contracts are deployed in distinct shards, meaning that the *state, logic and execution of a smart contract are maintained in only one shard, and the smart contracts maintained by different shards are isolated* from each other [29]. This causes *two performance problems.* ***First***, processing smart contract transactions requires complex multi-round consensus protocols (Figure 1a, Phase 2), which reduces the efficiency of smart contract execution

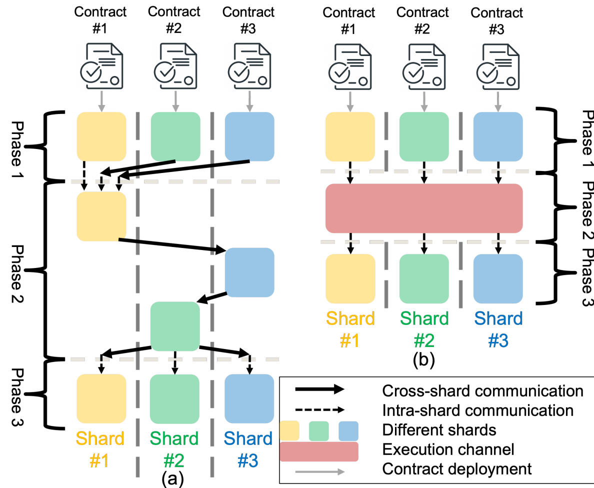


Fig. 1: Illustration for the basic procedure of contract transaction processing. (a): procedure in existing systems (e.g., [23]). (b): procedure in Jenga.

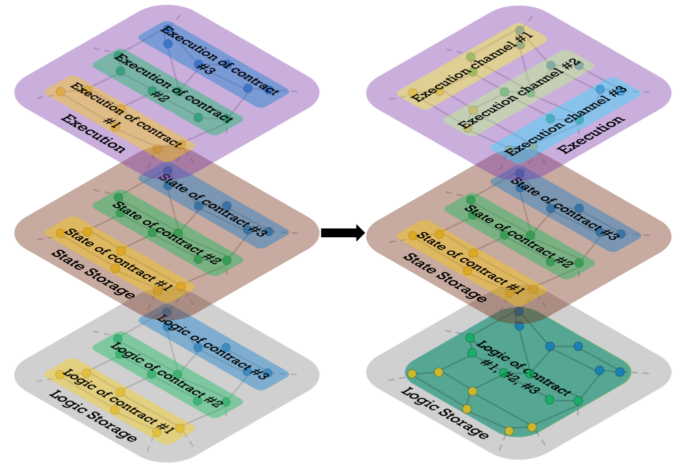


Fig. 2: Illustration for our intuition and main ideas. Left: in traditional blockchain sharding, different contracts are deployed to distinct shards as an entity, a contract's state, logic and execution is maintained by one shard. Right: we deconstruct the smart contract deployment and restructure the state, logic and execution for smart contracts.

(e.g., more than 50% throughput degradation in [13]). A smart contract transaction usually invokes multiple interdependent smart contracts, and the execution logic of these smart contracts is usually maintained separately on different shards. Therefore, to commit a smart contract transaction, related shards need to perform multiple rounds of contract execution, consensus, and cross-shard intermediate result delivery, which is extremely inefficient. ***Second***, due to the isolation between shards, processing a smart contract transaction usually requires fetching and returning multiple related states on different shards [27], [29] (Figure 1a, Phase 1, 3. It requires *both* cross-shard communication and intra-shard message broadcast). This process involves a lot of cross-shard communication, reducing the efficiency of state acquisition and return (e.g., cross-shard communication reduces 25% of transaction throughput in [21]).

To enable efficient smart contract processing in blockchain sharding, following challenges need to be addressed. First, how to reduce the performance loss caused by *multi-round cross-shard consensus protocols* during the execution of smart contracts. Second, how to reduce the performance loss caused by *cross-shard communication* during state acquisition and return. Moreover, while solving the above two challenges, it is necessary to maintain each node with limited storage and computation overhead for better scalability. We address the above challenges via orchestrating the smart contracts' logic storage, state storage and execution in blockchain sharding, as will be explained next.

Orchestrating Logic Storage. To address the first challenge, we require ***each shard to maintain the execution logic of all contracts*** (Figure 2, right, lowest layer). Under such design, multiple contracts can be executed together in one shard, needless for multi-round cross-shard executions. As a result, once a certain shard obtains the relevant states of a contract transaction, all relevant contracts can be executed within the same shard *in one round* (Figure 1b, Phase 2). To ease the storage overhead for each node, ***the states of a contract is solitarily stored in only one shard*** (Figure 2, right, middle layer). Since the logic storage overhead is not large (see Section VII-F), our system greatly reduces the performance loss caused by multi-round cross-shard consensus without introducing excessive storage burden.

Orchestrating State Storage and Execution. To address the second challenge, we set up ***multiple contract execution channels "orthogonally" across the shards storing isolated states*** (aka state shards) (Figure 2, right, upper 2 layers). Under such design, any execution channel can overlap all state shards. As a result, no extra cross-shard communication is required to transmit messages between any state shard and execution channel. Specifically, each node simultaneously belongs to a certain state shard and an "orthogonal" execution channel. The states obtained in multiple state shards can be directly broadcast to the corresponding execution channel through certain overlapped subgroups of nodes. After executing the contract, the nodes in the channel directly return the results to the relevant state shards through corresponding overlapped subgroups. This design *eliminates the cross-shard communications during state acquisition and return*, improving the system efficiency (Figure 1b, Phase 1,3. It requires *only* intra-shard broadcast).

With the above challenges addressed, we propose our system, named Jenga. Instead of treating contract as an indivisible entity (as do all the works mentioned above, see Figure 2, left), Jenga orchestrates the state storage, logic storage, and execution of smart contracts (Figure 2, right). To simplify the multi-round cross-shard consensus during contract execution, all contracts' *logic are copied to all shards.* To ease the

storage overhead on each node, *different shards store distinct states.* To eliminate the cross-shard communication during state acquisition and return, *nodes in different "orthogonal" channels execute contracts.* Based on these designs, Jenga increases system throughput and reduces transaction confirmation latency without too much storage overhead.

We implement a prototype of Jenga and several other baselines. We conduct extensive and large-scale experiments in Amazon EC2. Experimental results show that Jenga achieves significant performance gain compared with several state-of-the-art sharding based protocols. Compared with the most recent work [13], Jenga improves 1.5 times of system throughput and saves 65.2% per-node storage overhead at a large network size of 2880 nodes.

II. Background and Motivation

A. Blockchain and Smart Contract

Blockchain has drawn increasingly attentions from both research and industry areas [20], [28]. The proposal of Ethereum [28] has made the dissemination of smart contracts, thus expanding the application scenarios of blockchain to a new level. A smart contract is a self-executing contract with the terms of the agreement being directly written into lines of code [12]. A smart contract can be deployed in blockchain by its contract creator. When a smart contract is deployed, each node in the blockchain stores its execution *logic (aka functions)* and *states (aka data)*. Accounts (aka clients) interacting with the blockchain can invoke the deployed contracts by initiating a smart contract transaction. The nodes in the blockchain *obtain the account and contract states involved in the transaction, execute the relevant contract logic, and update the corresponding states.*

B. Sharding-Based Blockchain and Its Limitations

Traditional blockchain cannot scale its transaction processing capacity with the number of nodes in the network [3]. The main reason is that the consensus in traditional blockchain involves all nodes in the network, meaning each node needs to store and execute all transactions, and every consensus message needs to be broadcast in the whole network. It causes poor scalability in storage, computation and network, finally leading to unscalable system throughput [14], [17], [22].

One of the most promising approaches to increase blockchain throughput is to split the set of nodes into a number of smaller committees (aka shards) [17]. Each shard maintains a disjoint subset of states, executes different transactions in parallel, and reaches intra-shard consensus. Sharding has been an active research topic recently, both in industry [4], [7], [11], [23]–[25] and academia [9], [13]–[15], [17], [22], [29], [30].

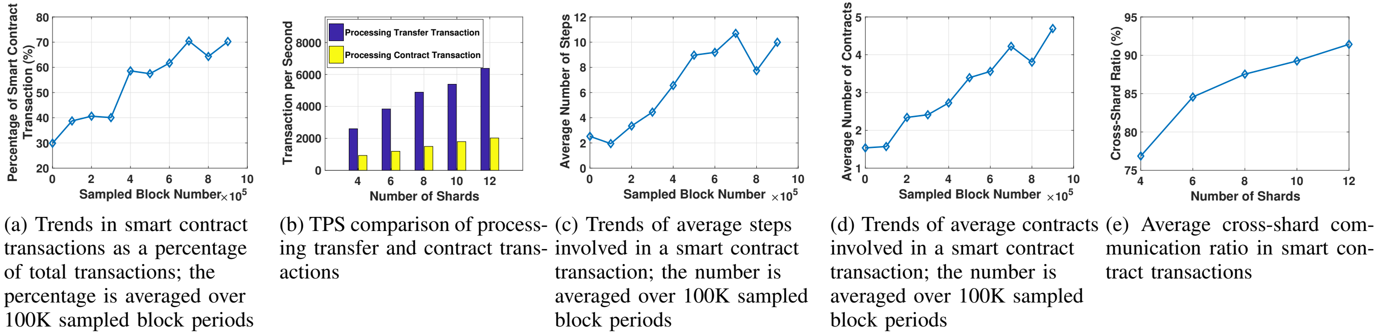
However, most of those works only focus on how to efficiently process the simplest kind of transactions (transfer of funds from one account to another), but ignore how to efficiently process smart contract transactions in blockchain sharding systems. Existing sharding systems have various drawbacks when tackling smart contracts, as will be introduced next.

C. Existing Solutions for Smart Contract in Sharding Systems

Only handful systems target at handling smart contracts in sharding-based blockchain, and their solutions suffer various problems. For example, in [4], [9], [25], all smart contracts are required to be processed in a specific shard. In such protocols, before initiating a smart contract transaction, the account needs to transfer its state to that specific shard, and transfer the state back after processing is completed. The scalability of this kind of design is poor, as the system's ability to process smart contracts will not increase as the number of shards increases. In [2], [7], [22], smart contracts can be deployed in different shards, but their designs only support processing smart contract transactions with single intermediate step (e.g., calling one function of one smart contract). However, a contract transaction usually consists of multiple steps (detailed explained in Section II-D), limiting the application scenarios of their designs.

The Cross-Shard Function Call [23] proposed by Ethereum supports the processing of smart contract transactions with multiple steps (Figure 1, left). Different smart contracts are randomly deployed in distinct shards. Each shard stores the state and logic of different smart contracts in isolation, and executes distinct smart contracts. For example, if smart contract A is deployed on shard 1, then only shard 1 has the state and logic of contract A, and contract A can only be executed on shard 1. When processing smart contract transactions, each transaction will be split into multiple sub-transactions. Different shards execute their related sub-transactions in the order of execution, reach a consensus on the execution results, and transmit the sub-transaction execution results to the shard responsible for executing the next sub-transaction. After all the sub-transactions are executed, the contract transaction is considered to be successfully processed, and it can finally be committed into the block. This design achieves high scalability in terms of storage and computation. However, before a smart contract transaction can be committed, their protocol requires multiple rounds of consensus and multi-step cross-shard communication to validate and process all related sub-transactions. This seriously degrades the sharding performance in terms of throughput and confirmation latency.

A most recent work aiming at improving the above solution is Pyramid [13]. It allows shards to merge, so that some nodes store the information (i.e., state and logic) of multiple shards and execute transactions for multiple shards. For some smart contract transactions, nodes in the merged shards can verify and process them directly. After processing the transactions in the merged shards, one more round of cross-shard consensus is required to commit the transactions. This protocol reduces multiple rounds of cross-shard consensus when processing smart contract transactions. However, as the nodes are working on multiple shards, their design imposes huge storage and computation overhead to the nodes, which weakens the sharding scalability. Moreover, in their design, only part of the shards are merged, meaning that there are still certain amount of contract transactions require multiple rounds of cross-shard



(a) Trends in smart contract transactions as a percentage of total transactions; the percentage is averaged over 100K sampled block periods

(b) TPS comparison of processing transfer and contract transactions

(c) Trends of average steps involved in a smart contract transaction; the number is averaged over 100K sampled block periods

(d) Trends of average contracts involved in a smart contract transaction; the number is averaged over 100K sampled block periods

(e) Average cross-shard communication ratio in smart contract transactions

Fig. 3: Measurement studies to motivate our work.

consensus to process.

Therefore, we aim to propose a sharding-based blockchain system for efficiently processing smart contract transactions with low overhead. To better motivate the design of our approach, we conduct measurement studies to show that (a) it is necessary to handle smart contract transactions efficiently, (b) how inefficient it is to process smart contract transactions in existing sharding-based blockchain, and (c) the possible reasons for the inefficiency of processing smart contract transactions in sharding-based blockchain.

D. Motivation

We conduct measurement studies based on real trace of Ethereum. We first demonstrate the need for efficient processing of smart contract transactions by showing the usage of smart contracts in Ethereum. We randomly sampled 1 million blocks (out of the most recent 10 million blocks) up to March 2021 [31]. According to the results of Figure 3a, in recent sampled blocks, smart contract transactions reached about 70% of total transactions. Moreover, the proportion of smart contract transactions in total transactions is showing an upward trend. The experiment in [22] also shows similar results. These results suggest that the use of smart contracts will continuously increase in the future, confirming that the need for efficient processing of smart contract transactions will become more urgent.

We now show how inefficient the existing sharding systems are when dealing with smart contracts. We implement a prototype of the Cross-Shard Function Call protocol. Figure 3b compares the system throughput at different system sizes for processing transfer transactions and smart contract transactions (see Section VII for evaluation details). As shown in the figure, the throughput in processing smart contract transactions is only around 1/3 of that in processing transfer transactions.

Finally, we clarify the possible reasons for the inefficiency of processing smart contract transactions in sharding-based blockchain. One of the main reason is that a smart contract transaction typically involves many smart contracts and contains multiple steps (e.g., in the train-hotel problem, a transaction may invoke a smart contract for booking a train ticket, followed by another smart contract for booking a hotel). Another reason is that the logic, state and execution of different smart contracts are solitarily maintained by distinct shards. As a result, processing smart contract transactions in blockchain sharding requires more complex multi-round consensus and extensive communication across shards than processing transfer transactions. Specifically, when processing a smart contract transaction, a significant amount of cross-shard communication is required to obtain (return) state information on multiple shards, before (after) executing smart contracts. More importantly, multiple rounds of cross-shard consensus are required to execute all the logic contained in a smart contract transaction.

We conduct experiments to justify that smart contract transactions typically contain multiple intermediate steps and smart contracts, and produce large amount of cross-shard communication. The results in Figure 3c and 3d show that in recent sampled blocks, the average number of steps and contracts involved in a smart contract transaction is 10 and 4.7, respectively. Moreover, the figures show an upward trend in the average number of steps and contracts involved in a smart contract transaction, indicating that more and more complex smart contract transactions are likely to be available in the future. Figure 3e shows the ratio of cross-shard communication in total communication when processing smart contract transactions under various number of shards. The results show that there are a large portion of cross-shard communication, and the number of cross-shard communication increases as the number of shards rises. In the case of 12 shards, the cross-shard ratio exceeds 90%.

III. System and Threat Model

A. System Model

In Jenga, there are N nodes, S state shards and S execution channels in the system. Like other systems [13], [15], [17], the nodes in Jenga are connected by a *partially synchronous peer-to-peer network* [8], in which there is a certain unknown upper bound Δ of message transmission delay in the network [16]. Similar to existing systems [13], [29], [30], each node has a unique public/secret key pair, given by a Public-Key Infrastructure (PKI). A public key represents the identity of a node. It will be broadcast through the network and recorded in an identity chain (as many existing sharding systems do, e.g., [13], [15]) once a node joins.

Jenga adopts the account model (similar to Ethereum) to represent the ledger state, in which each account (and smart contract) has its own states. The accounts send various transactions to the network, nodes in different shards record the states (e.g., balance) for different accounts and contracts. The states of a certain account or contract is maintained by only one shard.

B. Threat Model

There are two kinds of nodes in Jenga: honest and malicious. The honest nodes obey all the protocols in Jenga. However, malicious (Byzantine) nodes may deviate the protocols in arbitrary manners, such as sending arbitrary messages, sending messages with different values to different nodes, or failing to send any or all messages. The fraction of *total* malicious nodes in the system is denoted as f. In other words, fN nodes are controlled by Byzantine adversaries. Furthermore, similar to existing sharding systems [13], [29], [30], we assume that the Byzantine adversaries are slowly-adaptive, i.e., the set of malicious nodes and honest nodes are fixed during each epoch (e.g., one day) and can be changed only between epochs.

IV. System Overview

A. Objective

Our objective is to design a blockchain sharding system that efficiently handle smart contract transactions. The proposed system needs to not only support processing smart contract transactions with single step, but also handle more complex smart contract transactions efficiently. We design the system from the perspective of orchestrating smart contracts to break the isolation between shards. It simplifies the cross-shard communication and multi-round consensus during the smart contract transaction processing, thus improving the system performance. More importantly, the proposed system should not bring too much storage and computation overhead to the nodes, so as to ensure the sharding scalability.

To achieve the above goals, we propose two core designs. First, to simplify the multi-round cross-shard consensus during the execution of smart contracts, we propose a design that orchestrates the contract's logic storage. This design requires the execution logic of all smart contracts to be stored on all shards. Therefore, any shard can locally execute all smart contracts related to a transaction in one round. Second, to simplify the large amount of cross-shard communication that occurs when fetching (returning) state information before (after) executing smart contracts, we propose a design that orchestrates the contract's state storage and execution. We "orthogonalize" the state storage and execution of each shard, where any state shard can be directly connected to any execution channel through an overlapped subgroup of nodes. Thus, state information can be passed between the state shards and the execution channels directly via intra-shard broadcasts, without any extra cross-shard communication. We next describe the intuition for our designs and what the design entails.

B. Orchestrating Logic Storage

Intuition. In the existing sharding systems [11], [23], one contract is solitarily deployed to only one shard, and different smart contracts are deployed to distinct shards. As a result, shards do not know any information about each other's smart contracts. Therefore, a smart contract transaction needs to be split into several sub-transactions according to the smart contract information held by different shards. Relevant shards need to execute all the sub-transactions, conduct multiple rounds of consensus and transmit intermediate results in multiple rounds to commit a smart contract transaction, which is inefficient.

A straightforward idea to ameliorate the above problem is to require a shard maintain information about other shards [13]. This approach breaks the isolation between shards to a certain extent. Since a shard maintains more contracts, some smart contract transactions can be executed directly inside one shard without being split. However, for a shard maintains the information of n shards, the storage and computation overheads of the nodes inside it are n times higher. This causes huge storage and computation overhead for each node, reducing the sharding scalability.

Is there a smarter way to orchestrate contracts so that smart contract transactions can be processed efficiently without compromising the sharding scalability too much? We give an affirmative answer. Instead of treating each smart contract as an indivisible entity, we deconstruct its state storage, logic storage, and execution. When a smart contract is deployed, a node stores its logic (its function, code) and state (data). The first observation is that over time, *the amount of logic maintained by each node will be much less than the total maintained storage.* The reason is that nodes need to record huge amounts of smart contract transactions and frequently record state changes. In contrast, logic information only needs to be stored at deployment time and is not recorded as frequently. The second observation is that, *any shard can easily execute a smart contract if it has the logic of that contract.* Therefore, can we allow each node to maintain additional logic information, thus improving the smart contract process efficiency without adding too much storage overhead? Based on this intuition, we propose Network-Wide Logic Storage.

Network-Wide Logic Storage. In our design, when a smart contract is deployed, all shards store the logic for all contracts. However, to reduce per-node storage overhead, the state information of a contract is randomly (e.g., based on hash) stored by a particular shard. When a client sends a smart contract transaction, the shards associated with it locks the states required by that transaction and transmit the states to a particular shard (e.g., based on the transaction hash) for execution. Since each shard maintains the logic of all smart contracts, the execution shard after obtaining all relevant states can directly execute all smart contracts involved in that transaction, and return the execution results (state updates) to the relevant shards. Eventually, the relevant shards obtain the

states, unlocks them and commit the transaction to the block.

C. Orchestrating State Storage and Execution

Intuition. Another problem with existing solutions is that they require significant amount of cross-shard communication to fetch and return states on multiple shards during transaction processing. Specifically, a smart contract transaction usually requires multiple contracts' states located on multiple shards. When a shard processes the transaction, related states must be transmitted from other shards to it via additional cross-shard communication. Then the state will be broadcast within that shard for consensus.

Is there a way to transmit information between arbitrary shards without the extra cross-shard communication? We give a positive answer. We find that if the execution of contracts on one shard and the state storage on other shards are overlapped, then information between arbitrary state shards and execution shards can be transmitted directly through intra-shard broadcasts (instead of cross-shard communication then intra-shard broadcast). This eliminates the extra overhead caused by cross-shard communication. Based on this intuition, we propose Orthogonal Lattice Structure.

Orthogonal Lattice Structure. On top of the shards storing states (state shards), we orthogonally erect a corresponding number of execution channels. Each channel overlaps with all shards. State shards store the state of contracts, while execution channels execute contracts. Specifically, when a smart contract is deployed, its state information is randomly stored by a state shard. The execution of all smart contracts inside a smart contract transaction is randomly handled by one execution channel. A node belongs to both a state shard and an execution channel that overlaps with it. The set of nodes belonging to the same overlapped part forms a subgroup. When processing a smart contract transaction, the states on the state shards are broadcast directly to a particular execution channel via some overlapped subgroups of nodes. After the execution channel finishes the contract execution, the states are then broadcast back to the state shards via the overlapped subgroups. This design eliminates cross-shard communication for state fetch and return, and does not impose additional storage and computational overhead on the node.

V. System Design

A. Network-Wide Logic Storage

In Jenga, all shards store the logic of all contracts. A client who wants to create a smart contract can deploy the contract by initiating a contract-deploying transaction into the network. The contract-deploying transaction is broadcast to the entire network. Each shard internally verifies the contract-deploying transaction via intra-shard consensus and commit the transaction into a block. *Every shard needs to store the logic information of that smart contract, while only one shard needs to store its states.* Specifically, the states of a certain contract is randomly (e.g., based on the smart contract's hash) stored to a shard. Noting that different shards also maintain the states for different accounts, as all existing systems do.

B. Orthogonal Lattice Structure

In Jenga, each execution channel overlaps with all state shards. Different shards store distinct states, and different "orthogonal" execution channels process different smart contract transactions. When a contract is deployed, its state is randomly maintained by a certain shard (named state shard). However, this shard is not responsible for executing that smart contract. Since the logic of that smart contract is stored on all shards, any shard can execute it if the shard has that smart contract's state. So we assign the execution of the smart contract to one of the execution channels "orthogonal" to the state shard. Any two state shard and the execution channel are connected by a subgroup of nodes. That is, any single node exists on both a state shard and an execution channel orthogonal to it, and connects to the peers in that shard (channel). In this way, through a certain subgroup, any information between a state shard and an execution channel can be transmitted via intra-shard broadcast directly, rather than first passing the message from a shard to another through cross-shard communication then broadcast it in that shard. However, how to decide which node belongs to which execution channel and which smart contract should be executed by which execution channel in a decentralized system?

Determining the Execution Channel. To determine which execution channel a node should belong to, we use unbiased distributed randomness. There are many ways to generate distributed randomness, e.g, the verifiable random function (VRF) [19], verifable delay function (VDF) [5], and trusted execution environment [9]. It can be considered as a separated module in a sharding system and is orthogonal with our work, thus we do not discuss in detail. In our system, the distributed randomness determines both which state shard and execution channel a node belongs to. Specifically, each node i XOR its public key and the distributed randomness to get a new random number r_i. The new random number r_i is then modulo N to get a random number r_i^N. Each node divides r_i^N by the number of nodes that should be inside each shard (i.e., N/S), and the integer part of the result obtained is the state shard to which the node belongs. The random number r_i^N is divided by the number of shards (i.e., S) in the system, and the remainder of the result is the execution channel to which the node belongs. Such an allocation method implements an architecture in which the state shard is "orthogonal" to the execution channel. It also ensures that the number of state shards is the same as the number of execution channels, and the number of nodes inside each state shard is the same as the number of nodes inside each execution channel.

Another problem is that how to decide which execution channel a smart contract should be executed by. In existing systems, a specific smart contract can only be executed by a specific shard. However, in Jenga, each execution channel stores the logic of all contracts. Therefore, any execution channel can execute any smart contract. As a result, our system no longer uses the hash of a smart contract to determine the execution channel of a smart contract. Correspondingly, we

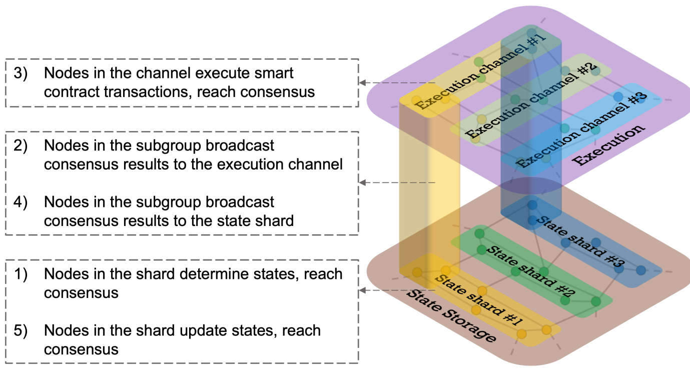


Fig. 4: Basic workflow of Jenga's cross-shard consensus. **Pre-prepare** contains procedure 1) and 2). **Prepare** contains procedure 3) and 4). **Commit** contains procedure 5).

use the *hash of the smart contract transaction* to determine the execution place of all smart contracts contained in it, and all smart contracts inside that transaction are executed by the same execution channel. Due to the randomness of the transaction hash, this design can better balance the computational load on each execution channel.

C. Cross-Shard Consensus Protocol

Based on the above design points, we now propose the cross-shard consensus protocol to commit smart contract transactions. A cross-shard consensus protocol is required in a blockchain sharding system, as processing a smart contract transaction usually involves several shards. The protocol eliminates the complex multi-round consensus during smart contract execution, as well as the extensive cross-shard communication during state fetching and return. Thus, our protocol effectively improves the efficiency of processing smart contract transactions in blockchain sharding systems. Our cross-shard consensus protocol consists of three phases: pre-prepare, prepare, and commit. The basic workflow of Jenga's cross-shard consensus is illustrated in Figure 4.

Phase 1: Pre-Prepare. In this phase, state shards determine the related states involved in a smart contract transaction via consensus, and broadcast them into the execution channel via subgroups. Cross-shard consensus starts with the client initiating a smart contract transaction (which invokes several smart contracts). Before sending the transaction, the client decides locally which contracts, accounts, and states are included in the smart contract transaction. This decision can be reached through dynamic program analysis (code simulation), similar to [23]. This information is then recorded inside the transaction. When sending the transaction, the client is required to pay amount of *transaction fee to prevent malicious behaviour (explained later).* The transaction is then sent to the shards where the relevant states are stored. Since the shard in which the smart contract state is stored is determined by the hash of the smart contract, any shard can easily verify whether it should handle a particular smart contract transaction. The relevant shard verifies the transaction via *intra-shard consensus* and *determines the state* maintained by its own shard with respect to that transaction.

Intra-Shard Consensus. We adopt Byzantine Fault Tolerant (BFT) consensus as our intra-shard consensus protocol (both in state shards and execution channels). To improve the consensus efficiency (both intra-shard and inter-shard), we adopt BLS aggregated signatures [6]. By aggregating the independent signatures generated by each node into one multi-signature, nodes only need to verify the correctness of the aggregated signature to determine whether the consensus result is valid or not. Such a design can be easily scaled to thousands of nodes.

State Determination. There are two outcomes of the state determination: (a), the state is available. In this case, the state shard reaches intra-shard consensus, sets the available state to unavailable and uses the transaction hash to determine which execution channel to send the state to. The state and its transaction hash are then broadcast directly in the execution channel through the subgroup of the state shard that overlaps with the execution channel. (b), the state is not available. In this case, the state shard reaches intra-shard consensus, broadcasts the *AbortRequest* message and the transaction hash to the execution channel through the subgroup. Since the execution channel is determined by the hash of a single transaction. Therefore, for a transaction, all the state information on different state shards will be broadcast to the same execution channel through different subgroups.

Transaction Fee. We leverage transaction fee to prevent clients from sending transactions indiscriminately. Without paying transaction fee, a client can intentionally send malicious transactions (e.g., a transaction containing wrong states to make the nodes lock wrong states). This damages the liveness of the system. Therefore, in Jenga, clients are required to pay a fee in advance when initiating a transaction. The amount of fee is related to the execution volume (similar to Ethereum's gas scheme [28]). With the fee scheme, if a malicious client sends a transaction that intentionally involves the wrong state or pays an insufficient fee, the node will find such an exception during the execution (e.g., a state that should be used in the contract is not included in the transaction). Once an exception is detected, the transaction is aborted and rolled back in all related shards, and all fees are deducted.

Phase 2: Prepare. In this phase, the execution channel executes all contracts related to a transaction, reaches intra-shard consensus, and broadcasts the execution results back to the state shards via different subgroups. A smart contract transaction is also sent to the execution channel that executes it. After the execution channel receives the result of the first state determination associated with a transaction, it starts a counter c for that transaction. c represents the number of states associated with the transaction, which can be derived by parsing the smart contract transaction. The counter is subtracted by one for each valid state successfully obtained. Eventually, if c=0, the execution channel starts executing all smart contracts within the transaction, reaches intra-shard

consensus, and returns the execution results (state updates) and the transaction hash directly to the corresponding state shards through different subgroups. Otherwise (e.g., due to timeout or invalid states), the execution channel returns the *Abort* information and transaction hash via subgroups directly to all the related state shards after reaching consensus.

Phase 3: Commit. In this phase, the state shards get execution results from the execution channel via subgroups, update the states via consensus, and commit the smart contract transaction. Each related state shard gets feedback on the execution channel from the corresponding subgroup. This information is broadcast directly into the shard via the subgroup. If a valid state update is received, the shard internally commits the transaction to the block via intra-shard consensus, and adds the block to the blockchain. At the same time, the shard update the stored state and restore the state to available. If a valid *Abort* message is received, the transaction is aborted.

D. Other Components

We now add brief remarks about the other components in the system beyond our main designs. First, the bootstrapping phase of the system is out of the scope of our paper. In real blockchain projects, the bootstrapping can be done in either a centralized [25] or decentralized [29] manner. Second, in order to prevent Sybil attack, Jenga uses PoS (Proof of Stake) as the access mechanism for nodes. Similarly, blockchain systems can also use PoW mechanism as an access threshold [13], [29]. Third, we use a combination of VRF and VDF to generate verifiable, unbiased, and unpredictable distributed randomness to assist the sharding reshuffle process (e.g., decide which node belongs to which shard/channel) in each epoch (typically 1 day) [13], [29]. Fourth, our system is orthogonal to the processing of traditional transfer transactions. The normal transfer transactions are processed still via the traditional cross-shard transaction processing scheme (e.g., [25], [27]).

VI. Security Analysis

We first analyze the failure probability for each shard and each subgroup. Based on the above analysis, we then calculate the failure probability of the system during each epoch. Under negligible system failure probability, we finally proof the safety and liveness for our cross-shard consensus protocol.

A. Shard Failure Probability

We first analyze the failure probability for each shard (i.e., fraction of malicious nodes in each shard is no less than 1/3) in each epoch, as the nodes will be reshuffled for each new epoch [9], [13], [29]. We do not analyze the reshuffle phase itself as it is orthogonal to our system. Noting that, in Jenga, the size for each shard and the number of nodes in each shard are all the same as those of a channel. Therefore, a channel can be considered as another form of a shard, and the following analysis can be applied to any channel as well.

Similar to previous works [9], [13], [29], we use the hypergeometric distribution function to calculate the failure probability of each shard. In particular, let X be a random variable representing the number of Byzantine nodes assigned to a shard (channel) of size $k = N/S$, given the overall network size of N nodes among which up to fN nodes are Byzantine. The upper bound of the probability for the shard failure in each epoch can be computed by:

$$p_{shard} = Pr[X \geq \lfloor k/3 \rfloor] = \sum_{x=\lfloor k/3 \rfloor}^{k} \frac{\binom{fN}{x}\binom{N-fN}{k-x}}{\binom{N}{k}}. \quad (1)$$

B. Subgroup Failure Probability

We now analyze the failure probability for each subgroup. Unlike the shard, the malicious nodes within each subgroup need not be limited to less than 1/3, since consensus is not required within the subgroup. The role of the subgroup is to broadcast consensus results between the state shards and the execution channels. Since a message sent by an honest node will eventually be received by any other honest node (Section III-A), it is only necessary to ensure that there is at least one honest node within each subgroup to ensure the security of the message broadcast.

The failure probability within each subgroup can also be calculated by hypergeometric distribution. Specifically, let Y be a random variable representing the number of Byzantine nodes in a subgroup of size $j = k/S$, given the shard size of $k = N/S$ nodes. Because the fraction of malicious nodes in each shard are restricted less than 1/3, there are up to $k/3$ Byzantine nodes in each shard. Therefore, the upper bound of the probability for the subgroup failure can be computed by:

$$p_{subgroup} = Pr[Y \geq \lfloor j \rfloor] = \sum_{y=\lfloor j \rfloor}^{j} \frac{\binom{k/3}{y}\binom{k-k/3}{j-y}}{\binom{k}{j}}. \quad (2)$$

C. Epoch Security

Similar to [29], the upper bound of the failure probability for the whole system in each epoch can now be calculated. The calculation should consider both the shard failure probability p_{shard} and the subgroup failure probality $p_{subgroup}$. Therefore, the upper bound of the system failure probability in each epoch is:

$$p_{system} = 2S \cdot p_{shard} + S^2 \cdot p_{subgroup}, \quad (3)$$

because there are S state shards, S execution channels, and S^2 subgroups in the system.

By carefully adjusting the number of nodes N and the number of shards (channels) S, we can bound the failure probability of the system to be negligible, as will be illustrated in Section VII-C.

D. Protocol Security Analysis

Under negligible system failure probability, our cross-shard consensus protocol achieves safety and liveness, as we will prove next.

Theorem 1 (Safety). *The cross-shard consensus achieves safety if there are no more than 1/3 fraction of malicious nodes in each shard (channel).*

Proof. Given no more than 1/3 malicious nodes in each shard (channel), the intra-shard BFT consensus (in all the three

phases) can be guaranteed as secure. Therefore, an intra-shard consensus result along with the BLS signature is honest. Meanwhile, the consensus result cannot be modified or forged because the signature can be used to detect forgery and tampering. Moreover, there exists at least one honest node in each subgroup. Therefore, the communication among shards and channels can be safely proceeded, which guarantees that all related shards (channels) can receive the valid consensus results. Moreover, the procedure of our cross-shard consensus is similar to the 2PC (two-phase commit) protocol in other distributed systems [2], [9], [13]. Therefore, honest nodes in all related shards and channels always agree on the same valid consensus results, i.e., our protocol guarantee atomicity and consistency for transactions. Therefore, the cross-shard consensus protocol achieves safety. □

Theorem 2 (Liveness)**.** *The cross-shard consensus achieves liveness if there are no more than 1/3 fraction of malicious nodes in each shard (channel).*

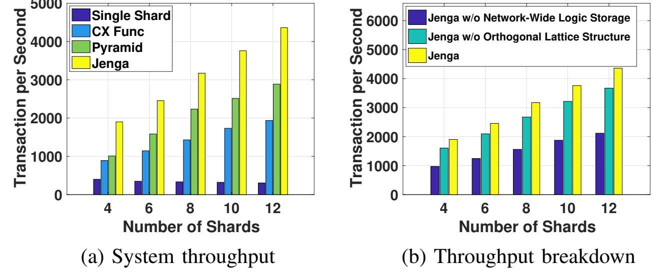
Proof. According to the system model in Section III-A, as the nodes are connected by a partially synchronous network and each shard (channel) has no more than 1/3 malicious nodes, the BFT protocol adopted as the intra-shard consensus of each shard (channel) can achieve liveness. According to Theorem 1, each involved shard always makes progress, and any transaction sent to the shards will eventually be committed or aborted. Such eventual availability means that no malicious node can block the consensus indefinitely. Therefore, the cross-shard consensus protocol achieves liveness. □

VII. Implementation and Evaluation

A. Implementation

We implement Jenga in Go language with 8,000+ LoC based on Harmony [25], a well-known blockchain sharding project within top 100 market cap in cryptocurrency. Harmony's implementation is based on Ethereum and it adopts a similar smart contract framework with Ethereum. However, Harmony only supports smart contracts to be deployed on one specific shard (see in Section II-C), while our implementation allows smart contracts to be deployed in individual shards, improving the scalability of the system. Since Ethereum is a widely adopted system and our system uses Ethereum as the underlying architecture, other Ethereum-based systems can easily apply our work to improve their throughput for smart contracts.

We also implement the prototype of three benchmark systems that support handling multi-step smart contracts for comparison, namely *Single Shard*, *CX Func*, and *Pyramid*. *Single Shard* represents those systems in which only one specific shard can handle smart contract transactions [4], [9], [25], *CX Func* represents the Cross-Shard Function Call proposed by Ethereum [23], and *Pyramid* is the Pyramid system proposed in [13]. Their main ideas are referred to in Section II-C. We implement their different cross-shard consensus of processing smart contract transactions. In addition, to ensure fairness of the comparison, we adopt the same intra-shard consensus protocol as Jenga in all the benchmark systems.



(a) System throughput (b) Throughput breakdown

Fig. 5: System throughput and throughput breakdown.

B. Evaluation Setup

We deploy Jenga on Amazon EC2 with up to 48 r5.24xlarge instances, each with a 96-core processor and a 25-Gbps communication link. We conduct evaluations at a scale of up to 12 shards and 2880 nodes. Similar to most running blockchain testbeds [13], [29], we consider a latency of 100 ms for every message and a bandwidth of 20 Mbps for each node. In a consensus round, each node can verify up to 4096 transactions, each of which is 512 bytes. We randomly sample 1 million smart contract transactions in the most recent 1 million Ethereum blocks up to March 2021 [31] for evaluations. The fraction of total malicious nodes f is set as 20%, similar to previous works [15], [17]. We compare Jenga with *Single Shard*, *CX Func*, and *Pyramid*.

TABLE I: Choice of number of nodes per shard and corresponding failure probability.

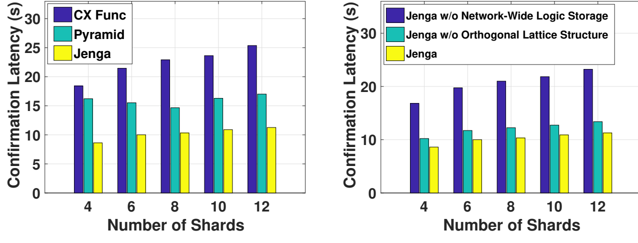
Number of Shards	4	6	8	10	12
# of Nodes per Shard	180	200	210	230	240
System Failure Probability ($\cdot 10^{-6}$)	1.6	6.1	5.1	5.3	2.8

C. Choice of Shard Size

We should adjust the shard size to limit the system failure probability to be negligible, as described in Section VI. The rule for selecting the shard size is: the failure probability is less than $2^{-17} \approx 7.6 \cdot 10^{-6}$ [13]. This failure probability guarantees that one failure will occur in about 359 years if the sharding system reshuffles in one-day epochs. We determine the number of nodes per shard based on Equation 1, 2 and 3. Table I shows the choice of shard size under different number of shards and the corresponding failure probability. Results show that our choice of shard size makes the probability of failure less than $7.6 \cdot 10^{-6}$ at any scale, ensuring the safety of the system. The following experiments will be conducted based on the shard size determined by Table I.

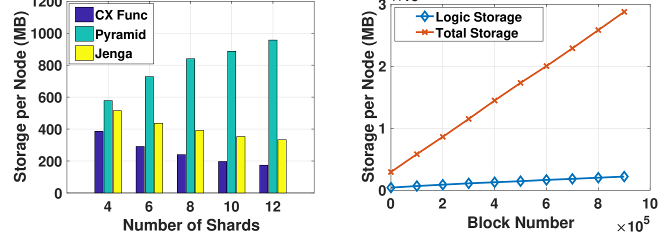
D. Throughput

We first evaluate the system throughput (i.e., TPS), the results are shown in Figure 5a. Compared to the baselines, Jenga achieves up to 14.3 times the throughput of *Single Shard* at a scale of 12 shards. And since *Single Shard* is



(a) Confirmation latency (b) Latency breakdown

Fig. 6: Confirmation latency and latency breakdown.



(a) Storage per node (b) Storage breakdown

Fig. 7: Storage overhead and storage breakdown.

not scalable, Jenga can be expected to perform even better than it as the number of shards increases. Moreover, Jenga reaches at most 2.3 times the throughput of *CX Func* and reaches 1.5 times the throughput of *Pyramid* at a scale of 12 shards. The performance improvement compared to *CX Func* is mainly because Jenga does not require multiple rounds of cross-shard consensus and large amount of cross-shard communication. Jenga outperforms *Pyramid* because of the following two reasons: first, Jenga does not require cross-shard communication; second, in Pyramid, merged shards cannot cover all transactions, there are still amount of transactions that need to be processed via multi-round cross-shard consensus. Jenga scales throughput at most 1.8 times when doubling the shard number (e.g., 6 to 12 shards), indicating that Jenga has good throughput scalability. Due to the poor performance of *Single Shard*, we will no longer compare Jenga with *Single Shard* in our subsequent experiments.

Breakdown. We breakdown the system to see how our main design points affect throughput gains. As shown in Figure 5b, the design of Network-Wide Logic Storage contributes more throughput gain. With logic of the smart contract deployed network-wide, Jenga compacts multiple rounds of cross-shard execution when processing transactions into single step, which gives up to 2.1× throughput gain. Orthogonal Lattice Structure brings about 1.2× throughput gain by eliminating cross-shard communication.

E. Confirmation Latency

Figure 6a shows the comparative results of the transaction confirmation latency (the delay between the time that a transaction starts to be processed until the transaction is committed, similar to previous works [13], [29]). Compared to *CX Func* and *Pyramid*, Jenga reduces latency by up to 55.6% and 33.8% at a scale of 12 shards, respectively. As the number of shards increases, the confirmation latency increases as well. This is mainly because that as the number of shards increases, the number of nodes inside each shard also increases, so the intra-shard consensus takes longer time to finish. Another reason is that with a higher number of shards, the processing of transactions involves more shards, thus causing more cross-shard communication.

Breakdown. Figure 6b shows the improvement in transaction confirmation latency brought by our main design points. Specifically, at a scale of 12 shards, the Network-Wide Logic Storage design reduces the transaction confirmation latency by 51.5%. The reduction in latency is mainly due to the design of Network-Wide Logic Storage reducing the latency caused by multiple rounds of cross-shard consensus during execution. The Orthogonal Lattice Structure reduces confirmation latency by 15.8%, as the cross-shard communication during state fetching and return is eliminated.

Remarks on the Performance Improvement. Compared with Network-Wide Logic Storage which provides remarkable performance gain (both throughput and latency), the performance gain brought by Orthogonal Lattice Structure is not very significant. We speculate the main reason is that, to simplify the implementation, we use the client to relay cross-shard communication in baselines. This implementation reduces the cross-shard communication [9] in baseline systems, improving the baseline performance. However, it is not secure enough, as the clients can be malicious [9], [27], [29]. We presume that Jenga would deliver more performance gains if the baseline systems use more secure schemes for cross-shard communication (e.g., cross-shard broadcast [15], [17] or routing via nodes [18], [25], [29]). We consider such implementation and comparisons as our future work.

F. Storage Overhead

We compare the storage overhead per node and show the results in Figure 7a. As the number of shards grows, the average single-node storage overhead for Jenga and *CX Func* gradually decreases. This benefit is brought by the storage scalability of sharding. In contrast, the storage overhead per node in *Pyramid* increases. This is because, as the number of shards increases, *Pyramid* requires more nodes to work on larger merged shards. As a result, the average storage and computation overhead per node increases. Jenga reduces up to 65.2% average storage overhead per node compared to *Pyramid* at a scale of 12 shards, and it can reduce more with the system scales. Moreover, the average storage overhead per node in Jenga is only a small amount more than *CX Func* (less than 200MB). This part of the storage overhead is due to that each node shares all the contract logic.

Breakdown. As each node additionally stores the logic of all contracts in Jenga, we evaluate the overhead caused by logic storage and total storage. We collect all smart contract

transactions in the last 1 million blocks and the smart contract-deploying transactions associated with them, and evaluate the storage overhead per node in the unsharded case. The results are shown in Figure 7b. The meaning of each point in the figure is the average storage overhead per node over the last x sampled blocks. The results show two facts: First, logic storage accounts for a very small percentage of the total storage; Second, the percentage of logic storage decreases over time. The main reason for the above observations is that contracts are invoked frequently (i.e., recording state changes in blockchain) but not deployed frequently (i.e., recording logic). Therefore, the design of having each node store all logic storage is reasonable and its advantage of saving storage overhead will increase over time.

VIII. Conclusions

We present Jenga, a sharding-based blockchain to process smart contracts efficiently. At its core, Jenga breaks the isolation between shards via orchestrating the state storage, logic storage and execution of smart contracts. Jenga allows all nodes to share the logic for all contracts, different shards store distinct states, and the nodes in "orthogonal" channels executes contracts. As a result, the processing of smart contract transactions are handled efficiently, and with low overhead. To commit the smart contract transactions safely and efficiently, we propose a cross-shard consensus that can guarantee the atomicity and consistency of transactions. Finally, we implement Jenga and the evaluation results illustrate the superiority of our system. Specifically, compared with the most recent sharding protocol, Jenga improves the system throughput by 1.5 times in a setting with 2880 nodes, with 65.2% per-node storage overhead reduced.

Acknowledgement

This work was supported by National Natural Science Foundation of China (Grant No. 61701216), Guangdong Provincial Key Laboratory (Grant No. 2020B121201001), Guangdong Innovative and Entrepreneurial Research Team Program (Grant No. 2016ZT06G587) and the Shenzhen Sci-Tech Fund (Grant No. KYTDPT20181011104007, JCYJ20180507181527806).

References

[1] H. Adams. Uniswap whitepaper. *URL: https://hackmd. io/C-DvwDSfSxuh-Gd4WKE_ig*, 2020.

[2] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis. Chainspace: A sharded smart contracts platform. *arXiv preprint arXiv:1708.03778*, 2017.

[3] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis. Sok: Consensus in the age of blockchains. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 183–198, 2019.

[4] P. Barrett. Zilliqa technical whitepaper, 2017.

[5] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch. Verifiable delay functions. In *Annual international cryptology conference*, pages 757–788. Springer, 2018.

[6] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.

[7] V. Buterin. Ethereum yanking, 2018. https://ethresear.ch/t/cross-shard-contract-yanking/1450.

[8] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[9] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data*, pages 123–140, 2019.

[10] H. Duan, J. Li, S. Fan, Z. Lin, X. Wu, and W. Cai. Metaverse for social good: A university campus prototype. In *Proceedings of the 29th ACM International Conference on Multimedia*, pages 153–161, 2021.

[11] A. Elrond. Highly scalable public blockchain via adaptive state sharding and secure proof of stake, 2019.

[12] Ethereum. Ethereum smart contract, 2021. https://ethereum.org/en/developers/docs/smart-contracts/.

[13] Z. Hong, S. Guo, P. Li, and W. Chen. Pyramid: A layered sharding blockchain system. In *IEEE INFOCOM 2021-IEEE Conference on Computer Communications*. IEEE, 2021.

[14] C. Huang, Z. Wang, H. Chen, Q. Hu, Q. Zhang, W. Wang, and X. Guan. Repchain: A reputation-based secure, fast, and high incentive blockchain system via sharding. *IEEE Internet of Things Journal*, 8(6):4291–4304, 2020.

[15] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[16] Y. Liu, J. Liu, M. A. V. Salles, Z. Zhang, T. Li, B. Hu, F. Henglein, and R. Lu. Building blocks of sharding blockchain systems: Concepts, approaches, and open problems. *arXiv preprint arXiv:2102.13364*, 2021.

[17] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 17–30, 2016.

[18] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[19] S. Micali, M. Rabin, and S. Vadhan. Verifiable random functions. In *40th annual symposium on foundations of computer science (cat. No. 99CB37039)*, pages 120–130. IEEE, 1999.

[20] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

[21] L. N. Nguyen, T. D. Nguyen, T. N. Dinh, and M. T. Thai. Optchain: optimal transactions placement for scalable blockchain sharding. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 525–535. IEEE, 2019.

[22] G. Pîrlea, A. Kumar, and I. Sergey. Practical smart contract sharding with ownership and commutativity analysis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1327–1341, 2021.

[23] P. Robinson. Ethereum cross-shard function call, 2020. https://ethresear.ch/t/atomic-cross-shard-function-calls-using-system-events-live-parameter-checking-contract-locking/7114.

[24] A. Skidanov and I. Polosukhin. Nightshade: Near protocol sharding design. *URL: https://nearprotocol. com/downloads/Nightshade. pdf*, page 39, 2019.

[25] H. Team. Harmony: Technical whitepaper, 2018.

[26] S. Viswanathan and A. Shah. The scalability trilemma in blockchain. *Medium online*, 20, 2018.

[27] J. Wang and H. Wang. Monoxide: Scale out blockchains with asynchronous consensus zones. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 95–112, 2019.

[28] G. Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[29] M. Zamani, M. Movahedi, and M. Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 931–948, 2018.

[30] M. Zhang, J. Li, Z. Chen, H. Chen, and X. Deng. Cycledger: A scalable and secure parallel protocol for distributed ledger via sharding. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 358–367. IEEE, 2020.

[31] P. Zheng, Z. Zheng, J. Wu, and H.-N. Dai. Xblock-eth: Extracting and exploring blockchain data from ethereum. *IEEE Open Journal of the Computer Society*, 1:95–106, 2020.