BrokerChain: A Cross-Shard Blockchain Protocol for Account/Balance-based State Sharding

Huawei Huang*, Xiaowen Peng*, Jianzhou Zhan*, Shenyang Zhang*, Yue Lin*, Zibin Zheng* Song Guo†
*School of CSE, Sun Yat-sen University, Guangzhou, China. Email: huanghw28@mail.sysu.edu.cn
{pengxw3, zhanjzh, zhangshy46, liny237}@mail2.sysu.edu.cn, zhzibin@mail.sysu.edu.cn
†Department of Computing, The Hong Kong Polytechnic University. Email: song.guo@polyu.edu.hk

IEEE INFOCOM 2022 - IEEE Conference on Computer Communications | 978-1-6654-5822-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/INFOCOM48880.2022.9796859

***Abstract*—State-of-the-art blockchain sharding solutions, say Monoxide, can induce imbalanced transaction (TX) distributions among all blockchain shards due to their account deployment mechanisms. Imbalanced TX distributions then cause *hot shards*, in which the cross-shard TXs may experience an unlimited length of confirmation latency. Thus, how to address the hot-shard issue and how to reduce cross-shard TXs become significant challenges of blockchain state sharding. Through reviewing the related studies, we find that a cross-shard TX protocol that can achieve workload balance among all shards and simultaneously reduce the number of cross-shard TXs is still absent from the literature. To this end, we propose BrokerChain, which is a cross-shard blockchain protocol devised for the account/balance-based state sharding. Essentially, BrokerChain exploits fine-grained state partition and account segmentation. We also elaborate on how BrokerChain handles cross-shard TXs through broker accounts. The security issues and other properties of BrokerChain are analyzed substantially. Finally, we conduct comprehensive evaluations using both a cloud-based prototype and a transaction-driven simulator. The evaluation results show that BrokerChain outperforms other solutions in terms of system throughput, transaction confirmation latency, the queue size of transaction pool, and workload balance.**

I. Introduction

Sharding technique is viewed as a promising solution that can improve the scalability of blockchains [1]–[10]. The idea of sharding is to *divide* and *conquer* when facing all the transactions (TXs) submitted to a blockchain system. Instead of processing all TXs by all blockchain nodes as the conventional manner, sharding technique divides blockchain nodes into different smaller committees, each only has to handle a subset of TXs [1]. Thus, the transaction throughput can be boosted in theory.

There are mainly three types of sharding paradigms, i.e., network sharding, transaction sharding, and state sharding. Since *network sharding* divides the entire blockchain network into smaller committees, it is viewed as the foundation of the other two sharding paradigms. In each round of the conventional sharding protocol, after committee's formation, a number of disjoint sets of TXs are assigned to those committees under the policy of *transaction sharding*. Then, committees run a specific consensus protocol locally, such as Practical Byzantine Fault Tolerance (PBFT) [11], to achieve a local consensus towards the set of assigned TXs. During those three sharding techniques, *state sharding* is the most difficult one because it has to ensure that all states of a blockchain are amortized by all shards. Currently, the state sharding is mainly staying in the theory phase. Several representative sharding solutions have been proposed, such as Elastico [1], Omniledger [2], RapaidChain [4], and Monoxide [8]. Those sharding solutions are based on either *UTXO* (Unspent Transaction Output) or *account/balance* transaction model. For example, Elastico [1] exploits the UTXO model, while Monoxide [8] adopts the account/balance transaction model.

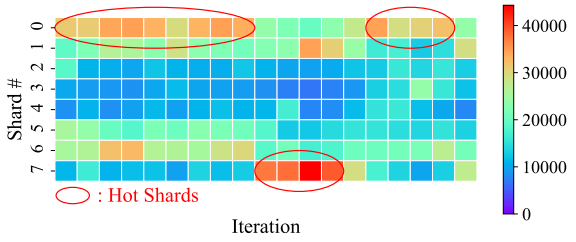


Fig. 1. When feeding 80000 TXs for each iteration of blockchain mining, Monoxide's sharding protocol [8] yields imbalanced transaction distributions among all shards. Consequently, some *hot shards* are induced. Here, we call a shard congested by an overwhelming number of TXs a *hot shard*.

Motivation. In Monoxide, user accounts are distributed to different blockchain shards according to the first few bits of their addresses, aiming to store all account states collaboratively. Although this manner can improve the system throughput, cross-shard TXs are inevitably induced. The handling of cross-shard TXs is a major technique issue under the account/balance-based blockchain sharding. Monoxide guarantees the atomicity of cross-shard TXs by introducing relay TXs. However, relay TXs may result in congested shards. We call such shards congested by an overwhelming number of TXs the *hot shards*. In each hot shard, TXs cannot be processed in time and may experience long confirmation latency. To prove our concern about Monoxide's sharding protocol, we evaluate the TX workloads of all shards by feeding 80000 TXs at each iteration of Monoxide's consensus. Fig. 1 shows the TX distribution results of Monoxide's sharding. We observe that hot shards widely exist in some shards during multiple iterations. In fact, these congested hot shards are caused by active accounts, which launch a large number of TXs frequently in their associated shards. Consequently, TXs are distributed over all shards in an imbalanced way. The insight behind those observations is that the account-deployment method adopted by Monoxide does not consider the frequency of launching TXs by accounts. Moreover, a great number of cross-shard

TXs could be also caused under Monoxide's sharding. As a result, when the number of shards becomes large, almost all TXs are cross-shard ones. When the receiver account of a cross-shard TX is suspending in a congested hot shard, Monoxide may incur infinite TX's confirmation latency.

Furthermore, through a thorough review over the state-of-the-art studies, e.g., Elastico [1], Omniledger [2], RapaidChain [4], we find that most of the cross-shard TX mechanisms mainly focus on the UTXO transaction model. Although Monoxide [8] and SkyChain [12] design their sharding protocols using the account/balance model, they didn't address the hot-shard issue. In summary, we have not yet found a cross-shard TX solution that can eliminate hot shards for the account/balance-based state sharding. Thus, we are motivated to devise such a new sharding protocol.

Challenges. When assigning a large number of TXs in the account/balance-based state sharding, a natural problem is how to reach load balance among all shards. Recall that the imbalanced workload in hot shards can cause large TX confirmation latency, which threatens the eventual atomicity of cross-shard TXs. This problem becomes even more challenging when most of the TXs are cross-shard ones [8]. Therefore, how to guarantee the eventual atomicity of cross-shard TXs becomes a challenge that prevents the state sharding mechanism from being largely adopted in practice.

To this end, this paper proposes a cross-shard blockchain protocol, named BrokerChain, for account/balance-based state sharding. BrokerChain aims to generate fewer cross-shard TXs and ensure workload balance for all blockchain shards. Our study in this paper leads to the following **contributions**.

- BrokerChain partitions account's state and performs account segmentation in the protocol layer. Thus, the well-partitioned account states can be amortized by multiple shards to achieve workload balance among all shards.
- To alleviate the hot-shard issue, BrokerChain also includes a cross-shard TX handling mechanism, which can guarantee the duration-limited eventual atomicity of cross-shard TXs.
- We implement BrokerChain protocol and deploy a prototype in Alibaba Cloud. Through both experiments and transaction-driven simulations, we show that BrokerChain outperforms state-of-the-art baselines in the perspectives of throughput, TX confirmation latency, the queue size of transaction pool, and workload balance.

The remaining of this paper is organized as follows. Section II reviews state-of-the-art studies. Section III describes the protocol design. Section IV depicts the handling of cross-shard TXs. Section V analyzes the security issues and other properties of BrokerChain. Section VI demonstrates the performance evaluation results. Finally, Section VII concludes this paper.

II. Preliminaries and Related Work

A. Transaction Models

There are two mainstream transaction models in existing blockchain systems, i.e., the UTXO model [13] and the account/balance model [14]. Under UTXO model, a TX may involve multiple inputs and outputs. When the outputs originating from previous UTXOs are imported into a TX, those UTXOs will be marked as *spent* and new UTXOs will be produced by this TX. Under the account/balance model, users may generate a TX using the deposits in their accounts. Ethereum [14] employs the account/balance model due to its simplicity that each TX has only one sender account and one receiver account. When confirming the legitimacy of a TX, it is necessary to check whether the deposits of the sender's account is sufficient or not.

B. Representative Blockchain Sharding Solutions

A great number of sharding solutions have been proposed to improve the scalability of blockchain systems [15]. Some of these representative solutions are reviewed as follows. Elastico [1] is viewed as the first sharding-based blockchain system, in which every shard processes TXs in parallel. Kokoris *et al.* [2] then present a state sharding protocol named OmniLedger, which adopts a scalable BFT-based consensus algorithm to improve TX throughput. Facing the high overhead of sharding reconfiguration in Elastico and OmniLedger, Zamani *et al.* [4] propose RapidChain to solve this overhead issue. Chainspace [3] exploits a particular distributed atomic commit protocol to support the sharding mechanism for smart contracts. Then, Wang *et al.* [8] propose Monoxide, which realizes an account/balance-based sharding blockchain. In Monoxide, a novel relay transaction mechanism is leveraged to process cross-shard TXs. Next, Nguyen *et al.* [5] present a new shard placement scheme named OptChain. This method can minimize the number of cross-shard TXs for the UTXO-based sharding. Prism [6] achieves optimal network throughput by deconstructing the blockchain structure into atomic functionalities. Dang *et al.* [7] present a scaling sharded blockchain that can improve the performance of consensus and shard formation. Recently, Tao *et al.* [9] propose a dynamic sharding system to improve the system throughput based on smart contracts. Huang *et al.* [10] propose an online stochastic-exploration algorithm to schedule most valuable committees for the large-scale sharding blockchain.

C. Cross-Shard Transaction Processing

A sharding blockchain must consider how to handle the cross-shard TXs. In Omniledger [2], the authors adopt a client-driven two-phase commit (2PC) mechanism with lock/unlock operations to ensure the atomicity of cross-shard TXs. Differently, Chainspace [3] introduces a client-driven BFT-based mechanism. RapidChain [4] transfers all involved UTXOs to the same shard by sub-transactions, such that a cross-shard TX can be transformed to intra-shard TXs. In Monoxide [8], the deduction operations and deposit operations are separated. Relay transactions are then exploited to achieve the eventual atomicity of cross-shard TXs. Recently, Pyramid [16] introduces a layered sharding consensus protocol. In this solution, cross-shard TXs are handled by a special type of nodes who serve two shards at the same time.

Comparing with these existing studies, the proposed BrokerChain handles cross-shard TXs by taking the advantages of broker accounts. Furthermore, BrokerChain can achieve the workload balance while distributing TXs through the fine-grained state-partition and account-segmentation mechanisms.

III. Protocol Design of BrokerChain

In this section, we present BrokerChain, an adaptive state sharding protocol based on account/balance transaction model.

A. Overview of BrokerChain

Similar to Rapidchain [4], we first define an *epoch* as a fixed length of system running time in BrokerChain. To avoid Sybil attacks [17], blockchain nodes should get their identities by solving a hash-based puzzle before joining in a network shard. In such puzzle, an unpredictable common randomness is created at the end of the previous epoch. Once a blockchain node solves the puzzle successfully, the last few bits of the solution indicate which shard the node should be designated to. In our design, BrokerChain consists of two types of shards:

- **M-shard.** A mining shard (shorten as *M-shard*) generates TX blocks by packing TXs and achieves the intra-shard consensus at the beginning of each epoch.
- **P-shard.** A partition shard (shorten as *P-shard*) is devised to partition account states in an adaptive manner during each epoch.

BrokerChain requires a number S of M-shards and one P-shard existing in the sharding blockchain. For both M-shard and P-shard, we adopt PBFT protocol [11] to achieve their respective intra-shard consensus and to avoid blockchain forks. Using Fig. 2, we now describe the most important four sequential phases of BrokerChain as follows.

- **TX-Block Consensus Phase.** At the beginning of an epoch, each M-shard packages TXs from the TX pool, and generates a number of TX blocks through running PBFT protocol. The number of TX blocks generated by a M-shard in an epoch depends on the intra-shard network parameters such as network bandwidth. Note that, the first new TX block generated by a M-shard follows the *state block* (i.e., B^{t-1} shown in Fig. 2), which records the state-partition results of the previous epoch.
- **State-Graph Partitioning Phase.** P-shard keeps loading the TXs from the new arriving blocks generated by M-shards, and keeps updating the state graph of all accounts. Once all TX blocks have been generated by M-shards within the current epoch, the state graph is fixed. Then, the protocol begins to partition the state graph, aiming to achieve workload balance among all shards.
- **State-Block Consensus Phase**. With the state graph of all accounts partitioned in previous phase, BrokerChain performs the *account segmentation*, which is described in detail in Section III-C. To reach a consensus towards the result of both state-graph partition and account segmentation, PBFT protocol is exploited again to generate a *state block* (denoted by B^t), which is then added to the P-shard chain.
- **State Reconfiguration Phase**. After reaching consensus on the partition result, P-shard broadcasts the state block B^t, which contains the S-partitioned new state graph, to all the associated M-shards. When receiving a state block B^t, a M-shard reads the state-partition result from the state block, and reconfigures its states accordingly such that TXs in the next epoch t+1 can be distributed to the designated shards according to the new account states.

At the end of each epoch, BrokerChain also needs to update the formation of both P-shard and M-shards. To update those shards, Cuckoo rule [18] is invoked such that the system can defend against the join-leave attacks [19], [20].

In the following, we elaborate the most critical operations and data structures in BrokerChain: *State-Graph Partition*, *Account Segmentation*, and the *modified Shard State Tree*.

B. State-Graph Partition

P-shard monitors the TX blocks generated by M-shards, and keeps loading the TXs in each arriving new block at real time to build/update the *state graph*. When receiving a specified minimum number of TX blocks, the state graph is fixed. As shown in the State-Graph Partitioning Phase of Fig. 2, each vertex of such state graph represents an account. The edge weight (denoted by w_e) is defined as the number of TXs that involve the corresponding pair of accounts, while the vertex weight (denoted by w_v) is calculated by the sum of the involved edges' weight. With the fixed state graph, P-shard begins to partition all accounts in the graph using Metis [21], which is a well-known heuristic graph-partitioning tool. Metis can partition the state graph into non-overlapping S parts, while reducing the number of cross-shard TXs and considering the workload balance among all shards.

C. Account Segmentation

In Monoxide [8], an account must be stored in only one single shard. The fact is that an account, such as an exchange's active account, may participate in a great number of TXs. This kind of active account would result in hot shards inevitably. Facing the hot-shard issue, we give an example in the following to show that this issue could be solved at the user layer. The only assumption required is that a user is allowed to hold multiple accounts, which are then permitted to store in multiple arbitrary state shards. Suppose that a specific user is holding multiple accounts. Following Monoxide's account-deployment rule, this user's accounts will be placed in different shards if the first k bits of account addresses are different. Then, this user can launch a large number of TXs through his multiple accounts such that those TXs can be distributed into designated shards. Consequently, the workload balance of all shards is possible to achieve, and the number of cross-shard TXs can be reduced as well, simultaneously.

Motivation to Account Segmentation. Users, however, generally do not have the motivation to open multiple accounts and deposit their tokens there. Because it's inconvenient to manage those distributed accounts. What's more, it does not make sense to enforce users to launch their TXs through

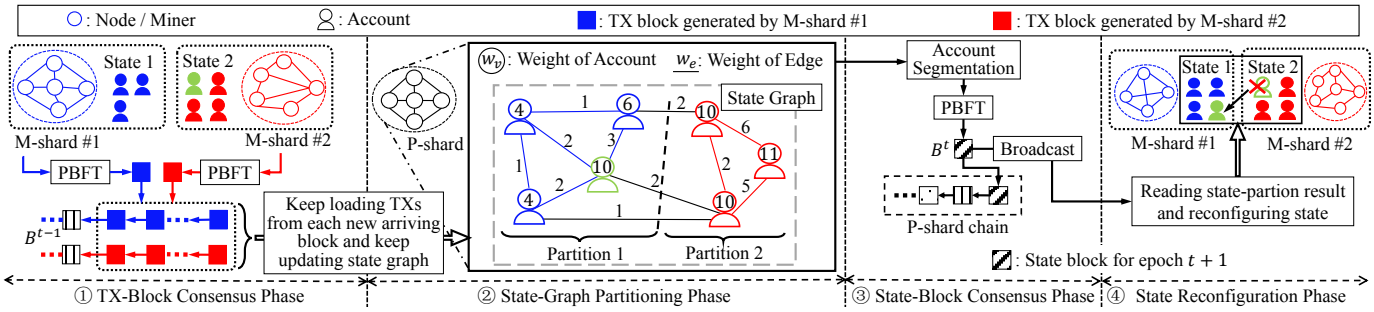


Fig. 2. Four major phases of BrokerChain protocol in an epoch t. There are two types of shards in BrokerChain: partition shards (P-shard) and mining shards (M-shard). In this figure, we only use two M-shards to illustrate the BrokerChain protocol.

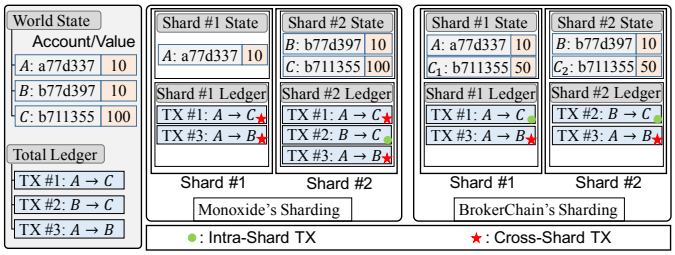


Fig. 3. Comparison of account segmentation results between Monoxide and BrokerChain. We see that BrokerChain yields fewer cross-shard TXs and a more balanced TX workloads over all shards.

their multiple accounts, aiming to balance the overall system workload. Therefore, we propose an *account segmentation* mechanism, which works in the protocol layer of blockchain architecture. The proposed account segmentation mechanism has the following advantages. i) Users are not enforced to open multiple accounts. ii) A user account's state can be easily divided and stored in multiple shards. iii) Through this user-transparent way, the workload balance of all shards is convenient to achieve such that hot shards can be eliminated.

Example of Account Segmentation. To better understand the proposed account segmentation mechanism, we use Fig. 3 to illustrate an example. Suppose that we have a sharding blockchain system with two shards (shard #1 and #2) and three accounts (denoted by A, B and C). Now the ledger includes 3 original transactions `TX:A→C`, `TX:B→C`, and `TX:A→B`. If the sharding system has already deployed accounts A and B in shard #1 and #2, respectively. Then account C can only be stored in shard #2 (or shard #1) under Monoxide's policy. As a result, the TXs "`TX:A→B`" and "`TX:A→C`" become 4 cross-shard TXs. In contrast, under BrokerChain, account C (with 100 tokens) can be segmented into two smaller accounts C_1 and C_2 (50 tokens for each), which are then placed to both shards, respectively. This account segmentation result is equivalent to that account C has 50 tokens stored in shard #1, and another 50 tokens stored in shard #2. Thus, the original TX "`TX:A→C`" turns to an intra-shard TXs. Furthermore, the workloads of both shards are balanced perfectly. From the example illustrated above, we see that the imbalanced TX workloads can be much alleviated and the number of cross-shard TXs can be reduced under BrokerChain.

The difference between the proposed account segmentation and the case where users autonomously store their deposits in multiple smaller accounts is that the addresses of the account segmented by BrokerChain are all identical. That is, BrokerChain protocol stores the deposits of an account located at different shards with the same account address. To identify an account's multiple states located at different shards, we adopt Ethereum's *counter* mechanism [14], which is called the *nonce* of an account's state.

D. Modified Shard State Tree

To enable the *state-graph partitioning* and *account segmentation* operations, BrokerChain has to enforce each shard to know the *storage map* of all accounts. Therefore, we devise a modified Shard State Tree (mSST) to store the account states based on Ethereum's *state tree* [14].

In contrast to the state tree, the BrokerChain's mSST is built on the storage map of all accounts. We denote such storage map as a vector $\mathbf{\Psi} = [e_1, e_2, ..., e_S]$ with each element e_i=0/1, $i \in \{1, 2, ..., S\}$, where S is the number of M-shards. Note that, BrokerChain needs to configure a storage map for every account. Only when e_i is equal to 1, can this specific account be viewed as being stored in shard i. If there are multiple elements in $\mathbf{\Psi}$ are labeled to 1, then the associated account is segmented to the same number of accounts, which are stored in each specific shard, respectively.

Data Structure of mSST. Fig. 4 illustrates the data structure design of mSST. The account state $\mathbb{S}_\mu$ of a specific user μ is represented by:

$$\mathbb{S}_\mu = \{X_\mu \mid \mathbf{\Psi}, \eta, \omega, \zeta\},$$

where X_μ denotes the account address of user μ, and η denotes the *nonce* field, which indicates the number of TXs sent from address X_μ or the contract creation operation generated by user μ. Then, ω denotes the *value* field, which shows the token deposits of the user. Finally, ζ denotes the *code* field, which represents the account type. Here, the account types include the user account and the smart contract account (the hash of the smart contract). To a specific account, different shards maintain different mSSTs for this target account. The difference is determined by the *value*, *nonce* and *code* fields of the local mSST. If a target account is not stored in shard any more, those fields corresponding to this account in this

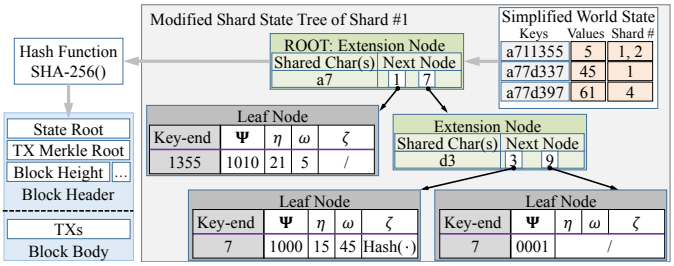


Fig. 4. Data structure of the devised modified *Shard State Tree* (mSST).

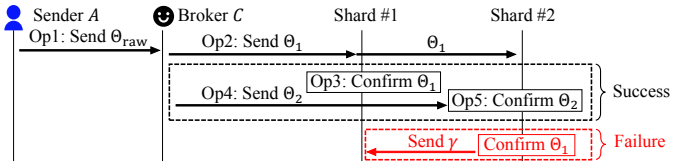


Fig. 5. The success and failure cases during the handling of cross-shard TXs. A success case includes all five operations: Op1-Op5, while a failure case typically indicates that Θ_2 is not received by shard #2 within shard #1's block-height interval $[H_{\text{current}}, H_{\text{current}} + H_{\text{lock}}/2]$.

shard's mSST will be removed. And this shard needs to only update the states for the target account. Any change of the target account's state will cause the change of state root of mSST. Therefore, it is easy to maintain the consistency of an account's states in the associated shard.

Query Complexity. Through querying the *storage map* in mSST, we can easily confirm whether a TX is an intra-shard TX or a cross-shard one. The querying time order is $O(1)$. Therefore, the protocol can easily get the deposits of an account by summing the *value* field of all its segmented account states. The deposit-querying time order is $O(\xi)$, where ξ is the number of shards where the account's segmented states are stored.

IV. Handling the Cross-Shard TXs

Although the proposed *account segmentation* mechanism enables an account to be segmented into multiple smaller ones, the question is that the blockchain system needs to recruit some accounts that are willing to act as the intermediaries to help handle cross-shard TXs. We call such intermediaries the *broker accounts*, which are shorten as *brokers*. In this section, we introduce how BrokerChain handles the cross-shard TXs through brokers. A fundamental requirement to become a broker is that its account has a sufficient amount of tokens. For the system users who intend to become brokers, they could request to pledge their assets to a trusted third party or a special smart contract. Then, BrokerChain protocol accommodates the broker eligibility for those interested system users. To stimulate a system user to play a broker's role, an incentive mechanism is necessary. The design of such incentive mechanism is left as our future work.

We still use the accounts and the ledger shown in Fig. 3 as an example to demonstrate the cross-shard TX handling. Suppose that C is a broker and its account is segmented and stored in shards #1 and #2. Now, we have a raw TX, i.e., sender A transfers a number v of tokens to the receiver B via broker C. Here, we call shard #1 the *source shard* and shard #2 the *destination shard* of this TX. Exploiting Fig. 5, we introduce both the success and failure cases of cross-shard TX handling.

A. Cross-Shard TX Atomicity Guarantee in A Success Case

The five operations of the successful cross-shard TX handling are described as follows.

- **Op1: Create a raw transaction Θ_{raw}.** Account A first chooses an appropriate token-lock duration (denoted by H_{lock}), which is in fact the number of successive blocks counting from the block including Θ_{raw} to the one releasing locked tokens. Θ_{raw} is defined as:
$$\Theta_{\text{raw}} := \langle\langle B, v, C, H_{\text{lock}}, \eta_{\text{sender}}, \eta_{\text{broker}}\rangle, \sigma_A\rangle,$$
where η_{broker} and η_{sender} denote the nonce of broker C and the nonce of sender account A, respectively. Symbol σ_A is the signature of sender A, calculated by invoking ECDSA algorithm [22]. The usage of the token-lock duration H_{lock} and nonces $\eta_{\text{sender}}, \eta_{\text{broker}}$ are explained subsequently. Next, A informs Θ_{raw} to the broker C.
- **Op2: Create the first-half cross-shard TX Θ_1.** When receiving Θ_{raw}, broker C creates the first-half cross-shard TX, denoted by Θ_1. Then, the current block height in shard #1 when Θ_1 is created is labeled as H_{current}. We then have
$$\Theta_1 := \langle\langle \text{Type1}, \Theta_{\text{raw}}, H_{\text{current}}\rangle, \sigma_C\rangle,$$
where Type1 is an indicator of Θ_1, and σ_C is the signature of broker C. Next, Θ_1 is broadcast to the blockchain network. When all the blockchain nodes in other shards receive Θ_1, they execute the following steps. i) Validating Θ_1 using signatures σ_A and σ_C. ii) Acquiring the sender's public key and calculating the the sender's account address using signature σ_A. iii) Through querying the mSST, shard nodes know the source and destination shards. iv) Using Kademlia [23] routing protocol, Θ_1 is forwarded to shards #1 and #2. After validating that η_{sender} is correct and the deposits of sender A are sufficient, Θ_1 will be added to the TX pool of shard #1, and waits to be packaged in a block.
- **Op3: Confirm Θ_1.** After broadcasting, the blockchain nodes in shard #1 shall include Θ_1 in the block whose height is labeled as H_{source}. Apparently, $H_{\text{current}} \leq H_{\text{source}}$. Then, sender A transfers v tokens to broker C, and these tokens will be locked within the block-height interval $[H_{\text{source}}, H_{\text{source}} + H_{\text{lock}}]$. The nonce of sender A will be increased by 1 to prevent the replay attack [24].
- **Op4: Create the second-half cross-shard TX Θ_2.** When broker C is acknowledged that Θ_1 has been confirmed, C then creates the second-half cross-shard TX Θ_2:
$$\Theta_2 := \langle\langle \text{Type2}, \Theta_{\text{raw}}\rangle, \sigma_C\rangle,$$
where Type2 is a label of Θ_2. Next, Θ_2 will be broadcast to the blockchain network. Θ_2 is finally routed to the

destination shard, i.e., shard #2. When η_{broker} is verified qualified and the deposits of broker C in the shard #2 are sufficient, Θ_2 will be added to the TX pool of shard #2.

- **Op5: Confirm** Θ_2. When Θ_2 is received by the blockchain nodes in shard #2, it will be packaged in a new block if the current block height of the shard #1 is smaller than $H_{\text{current}} + H_{\text{lock}}/2$. The account state in shard #2 is then updated when the following token transfer is executed: broker C transfers a number v of tokens to receiver B. Broker C's nonce is then increased by 1 to prevent the replay attack [24].

In the success case of cross-shard TX handling, receiver B can get a number v of tokens when Θ_2 is successfully included in the destination shard's block. On the other hand, only when the block height of shard #1 is greater than $H_{\text{source}}+H_{\text{lock}}$, can broker C receive the refund of v tokens.

B. Cross-Shard TX Atomicity Guarantee in A Failure Case

According to our design shown in Fig. 4, all shards mutually share their block headers where the essential block height information is included. Thus, if the blockchain nodes in shard #2 fail to receive Θ_2 when the current block height of shard #1 exceeds $H_{\text{current}} + H_{\text{lock}}/2$, we say that Θ_2 is failed to be acknowledged by shard #2. This failure result is very possibly induced by a malicious broker who intends to embezzle the sender account's locked tokens.

In order to process such failure case, BrokerChain executes the following steps to guarantee the atomicity of cross-shard TXs. In the first step, Θ_1 is included in a block located at shard #2 as shown in Fig. 5. Broker C's nonce will be attempted to update in shard #2. In the next step, blockchain nodes in shard #2 send the following confirmation of Θ_1 to shard #1 as the *failure proof* (denoted by γ):

$$\gamma := \langle \Theta_1, \mathit{dest}, H_{\text{dest}}, \{P_{\text{dest}}\} \rangle ,$$

where *dest* represents the index of the destination shard, H_{dest} denotes the height of destination shard's block where Θ_1 is included, and $\{P_{\text{dest}}\}$ refers to the Merkle tree path [13]. $\{P_{\text{dest}}\}$ consists of the hash values of all the associated Merkle tree nodes along the path originating from Merkle tree root and terminating at the entry hash node corresponding to Θ_1. The Merkle tree path $\{P_{\text{dest}}\}$ is used to verify that Θ_1 is packaged in a block located at shard #2. Once the nodes in shard #1 receive the failure proof γ and verify the correctness of path $\{P_{\text{dest}}\}$, shard #1 is acknowledged that Θ_1 has been included in a block located at shard #2 but Θ_2 was not. Then, γ will be included in the source shard's block whose block height is smaller than $H_{\text{source}} + H_{\text{lock}}$. Finally, the locked tokens in Op3 will be refunded to the sender account A.

No matter a cross-shard TX is handled in a success or failure case, only one of the two created TXs Θ_1 or Θ_2 is allowed to be included in a block at the destination shard. Note that, the failure proof γ is possibly lost by accident during broadcasting. Once γ is found lost, it can be reconstructed and launched again by the destination shard.

V. Property Analysis of BrokerChain

A. Tackling Double-Spending Attacks

BrokerChain treats the receiver of a cross-shard TX benign. Thus, in the following, we analyze how BrokerChain tackles typical security threats brought by the sender and broker.

Threat-1: Sender A is malicious. A malicious sender account may launch double-spending attacks. In such attacks, sender A may create a double-spending TX in the source shard along with the raw TX Θ_{raw} denoted by $\Theta_A := \langle A \rightarrow A', \eta_A \rangle$, where A' is sender A's another account and η_A is the nonce of Θ_A. Note that, to achieve double spending, η_A must be exactly same with η_{sender} signed in the firsts-half cross-shard TX Θ_1. Although the attack strategy described above seems feasible, it is impossible to implement under BrokerChain. Taking the advantage of the *counter* mechanism aforementioned, if Θ_A is confirmed by the source shard and η_A is updated, then Θ_1 will be failed to be confirmed by the source shard because its Θ_1's nonce η_{sender} is conflict to η_A. Only when broker C finds that Θ_1 has been confirmed by the source shard, C begins to create Θ_2. Through this way, broker C can defend against double-spending attacks launched by the malicious sender account.

Threat-2: Broker C is malicious. A malicious broker C may create the following double-spending TX in the destination shard right exactly when Θ_1 is confirmed in the source shard: $\Theta_C := \langle C \rightarrow C', \eta_C \rangle$, where C' is broker's another account and η_C is exactly same with η_{broker} signed in TX Θ_2. Similarly, if Θ_C is confirmed by the destination shard and the nonce of broker C is updated, then Θ_2 will not be confirmed by the destination shard because η_{broker} is conflict. By monitoring the update of nonce η_{broker}, BrokerChain can identify whether a broker is malicious or not. To prevent broker C from embezzling sender A's pledged tokens, BrokerChain enforces the handling routine of such cross-shard TX to fall into the failure case.

In conclusion, with the proposed asset pledge and token-lock mechanisms, BrokerChain can ensure the atomicity of cross-shard TXs under the threats of double-spending attacks.

B. Recommended Setting for Token-Lock Duration

Recall that in Op1 of cross-shard TX's handling described in Section IV-A, a sender needs to first choose an appropriate token-lock duration towards creating a raw transaction. The question is how to set such token-lock duration for the raw transaction. A feasible approach is to set the token-lock duration according to the system throughput observed currently. An empirical recommended setting is to ensure at least 20 times of the average latency of handling the cross-shard TXs.

C. Analysis of Cross-Shard Confirmation

As shown in Fig. 5, the cross-shard verifications occurred during the handling of cross-shard TXs include the following two cases: i) under the success case, broker C has to verify that whether Θ_1 is included in the source shard's chain, and ii) under the failure case, the source shard's nodes need to verify the *failure proof* γ sent by the destination shard.

Thus, in the success case, the computing complexity of cross-shard verification is $O(1)$. In the failure case, the computing complexity of cross-shard verification is $O(n)$, where n is the number of blockchain nodes in the source shard. Therefore, BrokerChain can lower such cross-shard verification overhead by shifting the on-chain verification to the off-chain manner taking advantages of the broker's role.

Regarding the cross-shard TX's confirmation latency, it is theoretically longer than twice intra-shard TX's confirmation latency. This is because a cross-shard TX must be confirmed at the source and destination shards in a sequential order. In contrast, BrokerChain can lower the confirmation latency based on broker's reputation. If a broker is completely trusted by the sharding blockchain, both Θ_1 and Θ_2 can be allowed to execute simultaneously. Thus, their confirmation latency can approach to that of an intra-shard TX technically.

D. Duration-Limited Eventual Atomicity

Monoxide [8] guarantees the *eventual atomicity* of cross-shard TXs. However, Monoxide does not specify the confirmation time of relay TXs. Thus, the cross-shard TX's handling in Monoxide may take a long time. In contrast, BrokerChain can ensure each cross-shard TX is done within the predefined token-lock duration H_{lock}. When the cross-shard TX is successfully processed, sender account's pledged tokens will be received by the receiver. Otherwise, sender's pledged tokens will be refunded by the broker. Therefore, we claim that BrokerChain ensures the *duration-limited* eventual atomicity for cross-shard TXs.

VI. Performance Evaluation

A. Settings

Experimental Prototype and TX-Driven Simulator: To evaluate the proposed BrokerChain, we implement both an experimental prototype and a TX-driven simulator. The prototype is implemented using Java and deployed in Alibaba Cloud. To study more comprehensive insights, we also implement a TX-driven sharding simulator using both Python and Java. We then run both the cloud-based prototype system and the sharding simulator by replaying collected historical TXs.

Dataset and its Usage: We use real Ethereum TXs as dataset, which contains 1.67 million historical TXs recorded from Aug. 7, 2015 to Feb. 13, 2016. At the beginning of each epoch, a number N_{TX} of TXs are prepared in the chronological order and replayed to the blockchain sharding system with certain arrival rates. Those TXs are then assigned to different M-shards according to their account's state.

Baselines: We consider the following three baselines. Monoxide [8] distributes accounts according to the first few bits of their addresses. LBF [25] updates the distribution of accounts periodically to achieve load-balanced TX distributions. Metis [21] purely emphasizes on the balanced partition on account's state graph.

Metrics: We first study the effect of several critical system parameters on various system metrics including TX throughput, TX's confirmation latency, and the queue size of TX pool. Regarding the workload performance, we then measure the total and the variance of shard workloads.

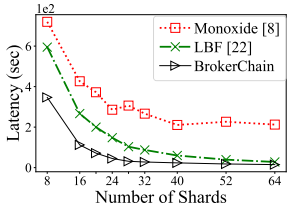
B. System Throughput and TX Confirmation Latency

First, we evaluate throughput and transaction confirmation latency using the cloud-based experimental prototype. We rent 112 virtual machines from Alibaba Cloud to deploy our prototype system, which includes 16 shards in total. Each virtual machine is equipped with 1 CPU core (Intel Xeon, 2.5/3.2GHz) and 2GB memory. The bandwidth of all network connections between nodes are set to 5 Mbps. For each M-shard, the block interval and block capacity are set to 8 seconds and 500 TXs, respectively. The TX arrival rate is fixed to 500 TXs/Sec. Since Metis has no correlation with TX's throughput and latency, we only compare the performance of BrokerChain with Monoxide and LBF. The experimental results are shown in Table I. We see that the average throughput of BrokerChain achieves 2.24× and 1.49× of Monoxide's and LBF's TPS, respectively. Furthermore, BrokerChain also maintains an average TX confirmation latency 275.94 seconds, which is much lower than that of the other two baselines.

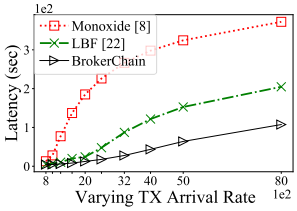
TABLE I
Experimental Results yielded by Cloud-based Prototype

Methods/Algorithms	Monoxide	LBF	BrokerChain
Avg. System throughput (TPS)	156.98	236.52	352.15
Avg. TX confirmation latency (sec)	1792.01	999.99	275.94

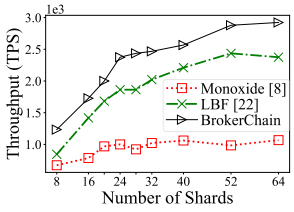
Next, we perform comprehensive TX-driven simulations by tuning more sophisticated system parameters. The shard block capacity and block interval is updated to 2000 TXs and 8 seconds, respectively. Through varying the TX arrival rate and the number of shards respectively, we still first study the average throughput and TX's confirmation latency under different sharding methods. The simulation results are shown in Fig. 6. Fig. 6(a) shows that when TX arrival rate is fixed to 3200, the average TX confirmation latency decreases when S increases from 8 to 32. This is because the arriving TXs in each shard overwhelm the shard's processing capacity when S is lower than 32. However, when S exceeds 32, the latency of BrokerChain converges under the arrival rate 3200. When S=64, the average TX latency of BrokerChain is as low as 14.87 seconds. Fig. 6(b) shows the performance of latency vs varying TX arrival rates, while fixing S=32. BrokerChain shows an overwhelming low latency comparing with other two baselines. This observation proves again that BrokerChain can guarantee the lowest latency while processing TXs even under a high TX arrival rate. Fig. 6(c) and Fig. 6(d) demonstrate the throughput vs S and the varying TX arrival rates, respectively. Although a large number of shards help increase the throughput under all methods, Fig. 6(c) shows that the benefit of shard number becomes saturated when S exceeds 52. This is because too many shards incur a large number of cross-shard TXs inherently. Those cross-shard TXs prevent the throughput from growing any further. Finally, Fig. 6(d) shows that BrokerChain has the largest throughput while varying the



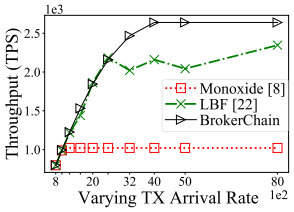
(a) TX arrival rate=3200 TXs/Sec



(b) S=32

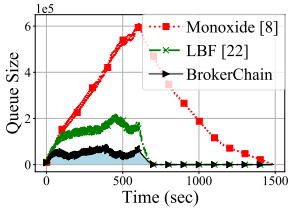


(c) TX arrival rate=3200 TXs/Sec

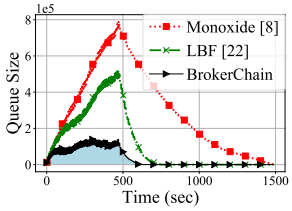


(d) S=32

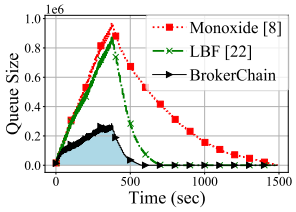
Fig. 6. Throughput and latency vs the # of shards and TX arrival rates.



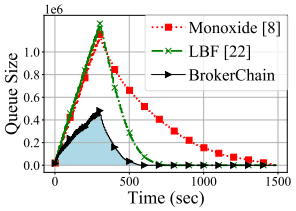
(a) TX arrival rate=2500 TXs/Sec



(b) TX arrival rate=3200 TXs/Sec



(c) TX arrival rate=4000 TXs/Sec



(d) TX arrival rate=5000 TXs/Sec

Fig. 7. Queue size of the TX pool.

TX arrival rate from 800 to 8000 and fixing S to 32. These results also indicate that BrokerChain's throughput becomes saturated when the TX arrival rate exceeds 4000, because the current settings hit the transaction processing capacity.

C. Queue Size of TX Pool

We then investigate the change of TX pool by monitoring its queue size. We use totally 1.67 million TXs to keep feeding the TX pool of the blockchain system with certain stable rates until all TXs are used up. Referring to the VISA's throughput, i.e., 4000 TPS approximately, we change the TX arrival rate within the range $\{2500, 3200, 4000, 5000\}$ TXs/Sec and fix K=40 for BrokerChain. The number of shards is fixed to 32. Fig. 7 shows that the queue size keeps growing when injecting TXs continually at the first few hundreds of seconds. When all 1.67 million TXs are consumed, the queue size shrinks. Among all methods, Monoxide maintains a linearly increasing queue and shows the largest. When the TX arrival rate is low, say 2500 TXs/Sec, LBF has a close queue size with BrokerChain. However, once the arrival rate increases, BrokerChain shows more capable to maintain a small TX pool than LBF and Monoxide. We attribute this result to the following insights. First, Monoxide tends to put TXs into a small number of hot shards. This policy induces a large number of cross-shard TXs, which cause large processing latency. Thus, the TX pool always has a large size under Monoxide. Second, LBF tries to distribute TXs evenly to all shards, thus the queue size of TX pool is maintained in a low level under a small TX arrival rate. However, LBF is not capable to handle the cross-shard TXs timely under a large arrival rate. In contrast, BrokerChain is capable to timely process all TXs in the TX pool.

D. The Effect of the Number of Segmented Accounts

Recall that broker's accounts can be segmented and deployed to different shards. Now we evaluate the effect of the number of segmented accounts by varying K and fixing S=64, N_{TX}=16e4. Through running 10 epochs, the workload

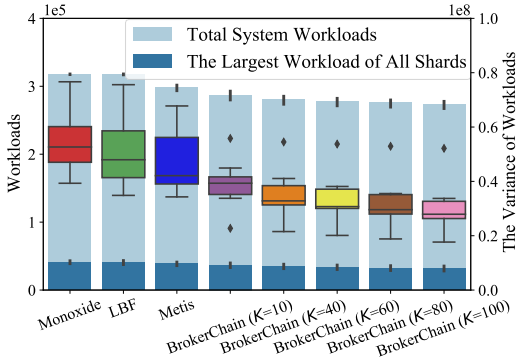
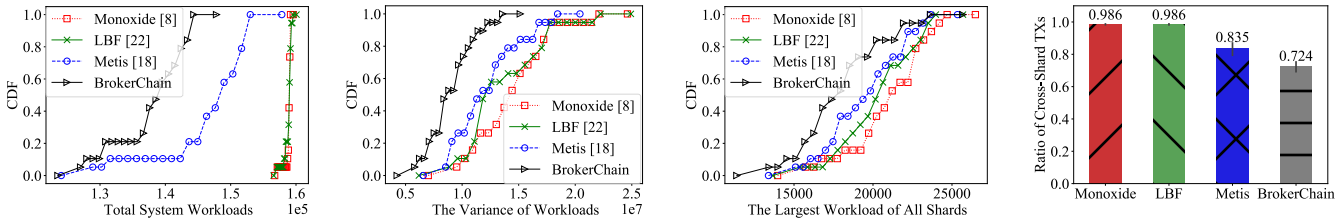


Fig. 8. Effect of the # of segmented accounts (i.e., K).

performance comparison is shown in Fig. 8, where the total system workload is the sum of all shard's TX workloads. We see that as K increases, the total system workloads, the largest and the variance of shard workloads all decrease. The reason is that with more segmented accounts, brokers can help reduce the number of cross-shard TXs and make the TX workloads more balanced than other baselines.

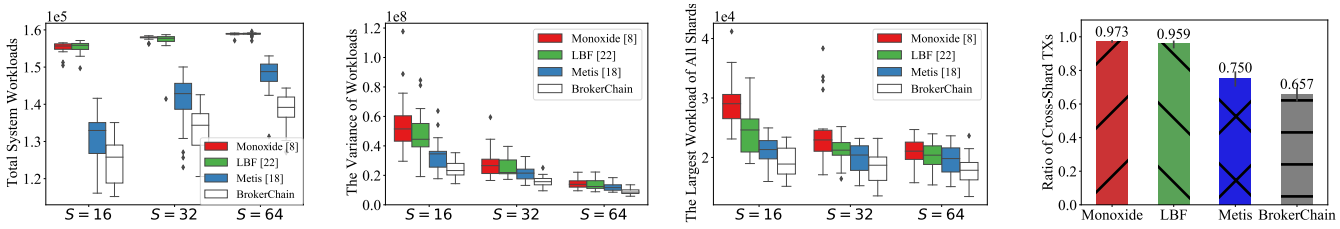
E. Performance of Shard Workloads

Through Fig. 9, we study the total, the largest and the variance of shard workloads under all methods. By assigning N_{TX}=8e4 to S=16 shards at each epoch and seting K=40 for BrokerChain, Fig. 9(a) shows that BrokerChain yields the lowest total system workloads. This is because BrokerChain can reduce the number of cross-shard TXs to a certain low degree. To have a clearer insight, we compare the ratios of cross-shard TXs in Fig. 9(d). The average cross-shard TX ratios of Monoxide, LBF, Metis and BrokerChain are 98.6%, 98.6%, 83.5% and 72.4%, respectively. Furthermore, we are also curious about the breakdown of total system workloads. Fig. 9(b) and Fig. 9(c) show that BrokerChain has the smaller variances of workloads and smaller the largest workload than that of other 3 baselines.



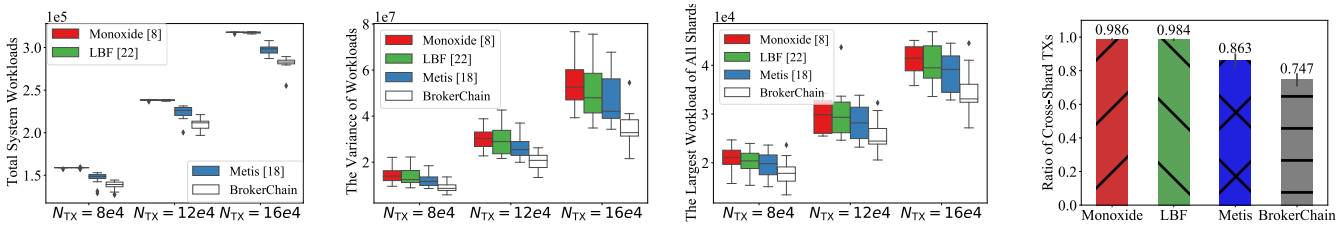
(a) CDF of total system workloads (b) CDF of the variance of workloads (c) CDF of the largest workload (d) Cross-shard TX ratio

Fig. 9. The workload performance under different methods, while fixing S=64, N_{TX}=8e4 and K=40. CDF stands for cumulative distribution function.



(a) Total system workloads vs S (b) The variance of workloads vs S (c) The largest workload vs S (d) Cross-shard TX ratio, S=32

Fig. 10. The effect of shard # on the total, the largest and the variance of workloads, while varying S within $\{16, 32, 64\}$, and fixing N_{TX}=8e4, K=40.



(a) Total system workloads vs N_{TX} (b) The variance of workloads vs N_{TX} (c) The largest workload vs N_{TX} (d) Cross-shard TX ratio, N_{TX}=12e4

Fig. 11. The effect of TX # on the total, the largest and the variance of workloads, while varying N_{TX} within {8e4, 12e4, 16e4}, and fixing S=64, K=40.

In the next group of simulation, to figure out the effect of the number of shards on workload performance, we vary S while fixing N_{TX}=8e4, K=40. As shown in Fig. 10(a), we see that the increasing S leads to growing total system workloads. This is because the number of cross-shard TXs increases following the increasing number of shards. Fig. 10(b) and Fig. 10(c) show that BrokerChain outperforms other baselines. Furthermore, BrokerChain also has the fewest *outliers* in figures. This observation implies that BrokerChain can make shard workloads more balanced and more stable than baselines. Again, Fig. 10(d) shows that BrokerChain still has the lowest cross-shard TX ratio.

Finally, to evaluate the effect of the number of TXs feeding per epoch on the workload performance, we vary N_{TX} within {8e4, 12e4, 16e4} and fix S=64 and K=40. Fig. 11(a) shows that BrokerChain maintains around 10%-20% lower the total system workloads than baselines. Although the increasing N_{TX} leads to growing system workloads, Fig. 11(b), Fig. 11(c) and Fig. 11(d) demonstrate that BrokerChain yields the lower variance, the lower largest shard workload, and the lower cross-shard ratio comparing with baselines.

VII. Conclusions

BrokerChain is proposed to serve as a cross-shard protocol for the account-based blockchain state sharding. In BrokerChain, the TX workload balance among all shards is achieved by the fine-grained state-graph partition and account segmentation mechanisms. BrokerChain handles the cross-shard TXs by exploiting broker accounts. The evaluation results obtained from both the cloud-based prototype and the TX-driven simulator demonstrate that BrokerChain outperforms the state-of-the-art sharding methods in terms of TX throughput, confirmation latency, queue size of TX pool, and the workload balance. In our future work, we plan to study the incentive mechanism that can inspire accounts to act as brokers.

Acknowledgment

This Work is partially supported by National Key R&D Program of China (No.2020YFB1006005), National Natural Science Foundation of China (61902445, 61872310), Guangdong Basic and Applied Basic Research Foundation (2019A1515011798), Guangzhou Basic and Applied Basic Research Foundation (202102020613), Hong Kong RGC Research Impact Fund (RIF) with the Project No. R5060-19, General Research Fund (GRF) with the Project No. 152221/19E, 152203/20E, and 152244/21E, Shenzhen Science and Technology Innovation Commission (R2020A045), and CCF-Huawei Populus euphratica forest fund (CCF-HuaweiBC2021004).

REFERENCES

[1] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A Secure Sharding Protocol For Open Blockchains," in *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS'16)*, 2016, pp. 17–30.

[2] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding," in *Proc. of IEEE Symposium on Security and Privacy (SP'18)*, 2018, pp. 583–598.

[3] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis, "Chainspace: A sharded smart contracts platform," in *Proc. of Network and Distributed System Security Symposium (NDSS'18)*, 2018.

[4] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," in *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*, 2018, pp. 931–948.

[5] L. N. Nguyen, T. D. Nguyen, T. N. Dinh, and M. T. Thai, "Optchain: optimal transactions placement for scalable blockchain sharding," in *Proc. of IEEE 39th International Conference on Distributed Computing Systems (ICDCS'19)*, 2019, pp. 525–535.

[6] V. Bagaria, S. Kannan, D. Tse, G. Fanti, and P. Viswanath, "Prism: Deconstructing the blockchain to approach physical limits," in *Proc. of the ACM SIGSAC Conference on Computer and Communications Security (CCS'19)*, 2019, pp. 585–602.

[7] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi, "Towards scaling blockchain systems via sharding," in *Proc. of the 2019 international conference on management of data (SIGMOD'19)*, 2019, pp. 123–140.

[8] J. Wang and H. Wang, "Monoxide: Scale out blockchains with asynchronous consensus zones," in *Proc. of 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 95–112.

[9] Y. Tao, B. Li, J. Jiang, H. C. Ng, C. Wang, and B. Li, "On sharding open blockchains with smart contracts," in *Proc. of IEEE 36th International Conference on Data Engineering (ICDE'20)*, 2020, pp. 1357–1368.

[10] H. Huang, Z. Huang, X. Peng, Z. Zheng, and S. Guo, "Mvcom: Scheduling most valuable committees for the large-scale sharded blockchain," in *Proc. of IEEE International Conference on Distributed Computing Systems (ICDCS'21)*, 2021.

[11] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *Proc. of Symposium on Operating Systems Design and Implementation (OSDI'99)*, vol. 99, no. 1999, 1999, pp. 173–186.

[12] J. Zhang, Z. Hong, X. Qiu, Y. Zhan, and W. Chen, "Skychain: A deep reinforcement learning-empowered dynamic blockchain sharding system," in *Proc. of 49th International Conference on Parallel Processing (ICPP'20)*, 2020, pp. 1–11.

[13] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Tech. Rep., 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[14] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[15] H. Huang, W. Kong, S. Zhou, Z. Zheng, and S. Guo, "A survey of state-of-the-art on blockchains: Theories, modelings, and tools," *ACM Comput. Surv.*, vol. 54, no. 2, mar 2021. [Online]. Available: https://doi.org/10.1145/3441692

[16] Z. Hong, S. Guo, P. Li, and W. Chen, "Pyramid: A layered sharding blockchain system," in *Proc. of IEEE Conference on Computer Communications, (INFOCOM'21)*, 2021.

[17] S. Zhang and J.-H. Lee, "Double-spending with a sybil attack in the bitcoin decentralized network," *IEEE transactions on Industrial Informatics*, vol. 15, no. 10, pp. 5715–5722, 2019.

[18] S. Sen and M. J. Freedman, "Commensal cuckoo: Secure group partitioning for large-scale services," *ACM SIGOPS Operating Systems Review*, vol. 46, no. 1, pp. 33–39, 2012.

[19] K. P. Puttaswamy, H. Zheng, and B. Y. Zhao, "Securing structured overlays against identity attacks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 10, pp. 1487–1498, 2008.

[20] B. Awerbuch and C. Scheideler, "Towards a scalable and robust dht," *Theory of Computing Systems*, vol. 45, no. 2, pp. 234–260, 2009.

[21] Karypis, George, Kumar, and Vipin, "A fast and high quality multilevel scheme for partitioning irregular graphs." *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[22] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)," *International Journal of Information Security*, vol. 1, no. 1, pp. 36–63, 2001.

[23] P. Maymounkov and D. M. Eres, "Kademlia: A peer-to-peer information system based on the xor metric," in *Revised Papers from the First International Workshop on Peer-to-peer Systems*, 2002, pp. 53–65.

[24] B. Hu, C. Zhou, Y.-C. Tian, Y. Qin, and X. Junping, "A collaborative intrusion detection approach using blockchain for multimicrogrid systems," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 49, no. 8, pp. 1720–1730, 2019.

[25] Z. Zhou, F. Li, H. Zhu, H. Xie, J. H. Abawajy, and M. U. Chowdhury, "An improved genetic algorithm using greedy strategy toward task scheduling optimization in cloud environments," *Neural Computing and Applications*, vol. 32, no. 6, pp. 1531–1541, 2020.