Account Migration across Blockchain Shards using Fine-tuned Lock Mechanism

Huawei Huang*, Yue Lin†, Zibin Zheng*‡
*SSE, Sun Yat-Sen University, China. Email: {huanghw28, zhzibin}@mail.sysu.edu.cn
†CSE, Sun Yat-Sen University, China. ‡Corresponding author.

***Abstract*—Sharding is one of the most promising techniques for improving blockchain scalability. In blockchain state sharding, *account migration* across shards is crucial to the low ratio of cross-shard transactions and cross-shard workload balance. Through reviewing state-of-the-art protocols proposed to reconfigure blockchain shards via account shuffling, we find that account migration plays a significant role. From the literature, we only find a related work that utilizes the *lock* mechanism to realize account migration. We call this method the *SOTA Lock*, in which both the target account's state and its associated transactions need to be locked when migrating this account between shards. Thereby, *SOTA Lock* causes a high makespan to the associated transactions. To address these challenges of account migration, we propose a dedicated *Fine-tuned Lock* protocol. Unlike *SOTA Lock*, *Fine-tuned Lock* enables real-time processing of the affected transactions during account migration. Thus, the makespan of associated transactions can be lowered. We implement *Fine-tuned Lock* protocol using an open-sourced blockchain testbed (i.e., BlockEmulator) and deploy it in Tencent cloud. The experimental results show that the proposed *Fine-tuned Lock* outperforms the *SOTA Lock* in terms of transaction makespan. For example, the transaction makespan of *Fine-tuned Lock* achieves around 30% the makespan of *SOTA Lock*.**

***Index Terms*—Blockchain, Sharding, Account Migration**

I. Introduction

Sharding is a popular technique for improving blockchain scalability while preserving the decentralization property of blockchains [1]–[13]. The basic idea of blockchain sharding is to divide all blockchain nodes into several groups, each called a network shard. All blockchain shards process transactions (TXs) in parallel. Thus, blockchain sharding is promising to improve blockchain throughput and reduce the makespan of transaction execution.

In *state sharding*, a challenge is that cross-shard TXs are generated when a TX involves accounts' state data stored in different shards. Even though various solutions [6]–[9] have been proposed to handle cross-shard TXs, the long makespan of transaction's confirmation and the high ratio of cross-shard TXs remain as significant obstacles that largely affect the throughput and scalability of a sharded blockchain.

Another challenge lies in the imbalanced TX workloads across blockchain shards. This results in some congested shards having a significant number of TXs queued in their local transaction pools, while others remain idle. Fig. 1 illustrates how an inappropriate account allocation can lead to imbalanced TX workloads. This issue can be addressed by implementing a fine-tuned account allocation mechanism.

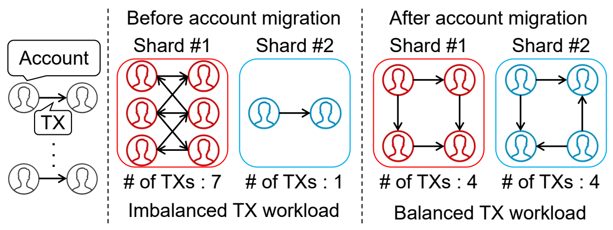


Fig. 1. Different account allocation solutions induce different workloads among blockchain shards.

Motivation. To reduce the number of cross-shard TXs while balancing workloads across shards, state-of-the-art studies [14]–[18] have paid their attention to shuffling a portion of accounts across blockchain shards. This account shuffling method aims to move those accounts that have frequent mutual interactions to the same shard such that the number of cross-shard TXs is possibly reduced. During those related SOTA studies, only LB-Chain [18] has implemented *account migration* across shards by exploiting the *lock* mechanism (we call *SOTA Lock* hereafter), which is illustrated in Fig. 2. However, the *SOTA Lock* mechanism has the following two drawbacks. Firstly, all the TXs of a specific account under migration have to be locked and congested in the source shard of this account. Thus, this SOTA Lock action definitely prolongs the makespan of the TXs associated with the target account. Secondly, no restoration solution has been proposed when migration failure occurs (e.g., migration times out).

For the first drawback aforementioned, a strawman method is to not lock any TXs related to the target account during its migration. Furthermore, the source shard keeps processing these TXs before the migration is completed. The destination shard can also process the TXs related to the target account under migration after receiving its associated state. However, as the two shards are not in perfect synchronization, the source shard may not immediately know when the migration was finished. Thus, both shards can execute these related TXs simultaneously. Consequently, there is a chance that both shards *deduct* more tokens from the target account that exceed its balance. Hence, this strawman method is susceptible to vulnerabilities in terms of account security.

In summary, although the SOTA studies have realized the significance of account migration, we have not found a practical implementation of *account migration* across blockchain shards in the context of blockchain state sharding.

Challenges to implementing account migration. When an account is no longer bonded with a shard, it needs to be migrated to a newly designated shard. Designing a protocol for account migration is not straightforward. Referring to the existing account migration solutions [18]–[21], migrating an account needs to lock both the account's state and its all associated TXs. This will cause a higher makespan to these TXs. Therefore, how to mitigate the impact of an account's migration on its associated TXs and guarantee the security of the blockchain system is a challenge.

To this end, we propose a fine-tuned cross-shard account migration protocol for state sharding. Our goals are three-fold, i.e., i) to implement a practical account migration protocol, ii) to minimize the impact of account migration on the associated TXs, and iii) to guarantee the liveness and consistency of a blockchain system while performing account migration.

Our study in this paper has the following **contributions**.

- **Originality.** We devised an account migration protocol using *fine-tuned Lock* mechanism, which can alleviate the impact of account migration on its associated TXs.
- **Methodology.** To enable account migration, we modified the data structures of both the ledger's `state` and `block`. We then modified the conventional *relay transaction* mechanism, aiming to address the system vulnerabilities that may arise from account migration.
- **Implementation** and **Usefulness.** Finally, we implemented our protocol on the open-sourced testbed BlockEmulator and deployed it to Tencent Cloud. Experimental results show that the proposed protocol outperforms two other account migration protocols in terms of TX makespan.

II. Related Work

A. Blockchain Sharding

Blockchain sharding is a technique that can be used to improve the scalability of a blockchain network by allowing it to process more TXs concurrently. Elastico [1] is viewed as the first sharded blockchain system. TXs are partitioned into different shards in Elastico and thus can be processed in parallel. However, every shard still stores the whole state ledger. To save storage resources, the authors of OmniLedger [6] proposed *state sharding*, in which each shard is responsible for storing and processing a subset of the ledger's data. A couple of other state-of-the-art *state sharding* protocols have been proposed, *e.g.,* Monoxide [9], Pyramid [22] and BrokerChain [15].

B. Processing Cross-shard Transactions

In a *state sharding* blockchain system, it is inevitable that there will be cross-shard TXs generated across shards. A cross-shard TX is a transaction that involves two or more shards and requires coordination among those shards. To handle cross-shard TXs, 2-Phase Commit (2PC)-based protocols are adopted by OmniLedger [6], RapidChain [7] and ChainSpace [8]. In addition, Monoxide [9] introduces *relay transaction* mechanism to achieve the *eventual atomicity* of

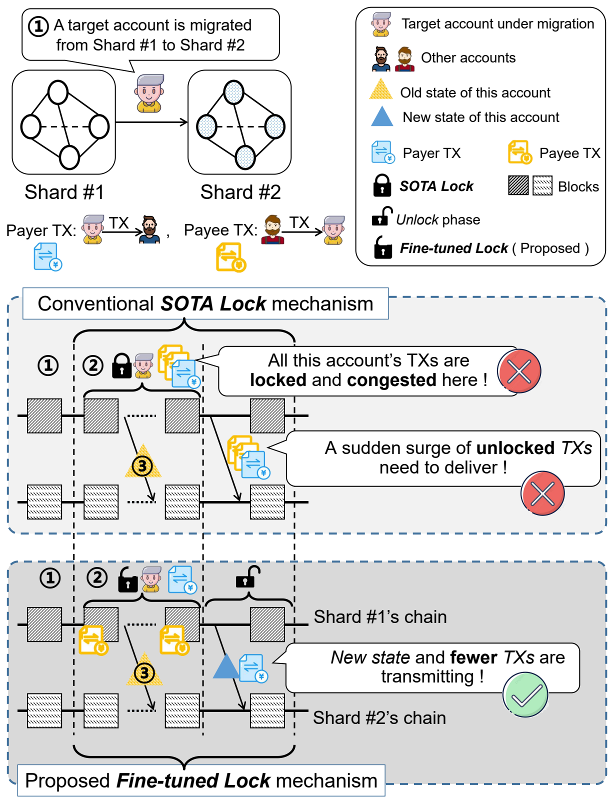

① A target account is migrated from Shard #1 to Shard #2
Shard #1
Shard #2
Payer TX:
TX
Payee TX:
TX
Target account under migration
Other accounts
Old state of this account
New state of this account
Payer TX
Payee TX
SOTA Lock
Blocks
Unlock phase
Fine-tuned Lock (Proposed)
Conventional SOTA Lock mechanism
①
②
③
All this account's TXs are locked and congested here !
A sudden surge of unlocked TXs need to deliver !
Shard #1's chain
New state and fewer TXs are transmitting !
Shard #2's chain
Proposed Fine-tuned Lock mechanism

Fig. 2. The comparison between *SOTA Lock* and the proposed *Fine-tuned Lock* mechanisms. ① : A migration is triggered. ② : Lock phase gets started. ③ : Sending *old state* of this account to Shard #2.

cross-shard TXs. In these mechanisms, the makespan of cross-shard TXs will be higher than that of intra-shard TXs. Furthermore, an excessively high proportion of cross-shard TXs increases the complexity of the sharded blockchain system, thereby affecting system performance, such as throughput and transactions' makespan.

C. Account Migration

Recently, to reduce the proportion of cross-shard TXs while balancing workloads across shards, some emerging studies are investigating *shard reconfiguration*. The reconfiguration refers to rescheduling accounts to appropriate shards in the account-based blockchain state sharding. Several studies attempt to migrate accounts, to reduce the proportion of cross-shard TXs while balancing the workload across shards. Several representatives are reviewed as follows. Shard Scheduler [14] designs an algorithm that can generate recommendations for account migration based on the interaction between accounts. An incentive mechanism is also provided in Shard Scheduler, to motivate the miners to follow the recommendations. BrokerChain [15] uses historical TXs to generate a global account graph. With such a global graph, all accounts are partitioned into appropriate shards in each epoch using Metis [23], which is a well-known heuristic graph-partitioning tool. Transformers [17] devises a community-aware account partition algorithm to make a trade-off between the ratio of cross-shard TXs and the workload balance. However, the approaches aforementioned only focus on determining which shards should accounts be migrated to, without practical implementation of *account migration*. Since account migration is crucial to shard reconfiguration, a practical solution of account migration is required to implement a blockchain sharding system.

Via a thorough review of the literature, we only found a *SOTA Lock* mechanism [18] was proposed to migrate accounts across blockchain shards. In more detail, before migrating an account to a new shard, the account's state should be locked at first. During the migration, no changes can be made to this account. Thus, any TX that modifies the target account under migration would be locked and would be migrated to the destination shard afterward. As a result, these TXs have to experience a long locking latency in the source shard, which prolongs the makespan of these TXs.

To the best of our knowledge, we have not found a study that investigates the impact of account migration on the TXs associated with the target accounts under migration. Compared with the state-of-the-art *SOTA Lock* mechanism, our proposed protocol enables real-time processing of a portion of those affected TXs. Thus, it can alleviate the large queueing latency of the TXs associated with the target account under migration.

III. Preliminaries and System Model

A. Preliminaries

The proposed protocol executes on top of the account-based transaction model. Similar to most sharding blockchains [22], [24]–[29], we focus on state sharding. In our system, each account's state is only managed by one shard at a time, *i.e.,* each shard manages a disjoint subset of all accounts. We assume that blockchain nodes in one shard can obtain the headers of a specific block committed in another shard. This is achieved through on-demand queries. Thus, each blockchain node can verify the validity of an account migrated from another shard. As for the intra-shard consensus, our system adopts Practical Byzantine Fault Tolerance (PBFT) [30] consensus. Furthermore, we exploit the *relay transaction* mechanism [9] to process cross-shard TXs.

B. Proposed Migration vs. SOTA Lock Migration

The proposed *account migration* method differs from the *SOTA Lock* and strawman methods mentioned earlier. It only locks the TXs that deduct money from the account (i.e., *payer TXs*) during the migration process. The *payee TXs*, in which the target account is payee, are not locked and can still be processed in the *source shard* as other regular TXs. Thus, we call our account migration the *Fine-tuned Lock* mechanism.

Before elaborating on our method in the next section, we use Fig. 2 to compare the proposed *Fine-tuned Lock* method with the *SOTA Lock* method. Suppose that an account is being migrated from Shard #1 to Shard #2. According to the conventional *SOTA Lock* approach, the account needs to be locked in Shard #1 and unlocked in Shard #2 when the migration is completed. All the TXs related to this account cannot be processed during the account's migration, and are locked in Shard #1, too. These TXs will be sent to Shard #2 after the migration is completed (during the *Unlock Phase*). In contrast, in our proposed *Fine-tuned Lock* method, only *payer TXs* are locked in the source shard. The unlocked *payee TXs* can still be processed in real time while the account is being migrated. Once the migration is completed, a new state of the target account together with its *payer TXs* are needed to deliver to Shard #2. As a result, the impact of account migration on the associated TXs is alleviated.

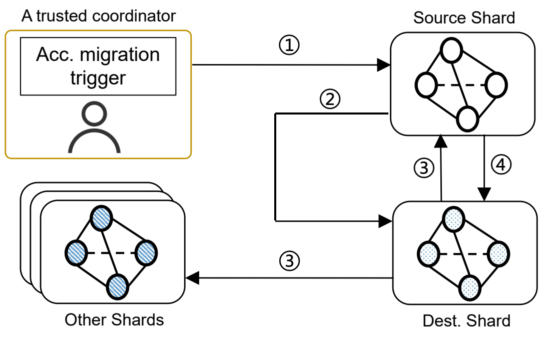


Fig. 3. A basic workflow of the proposed account migration protocol. ① : Requesting for migrating a target account from Source Shard to Dest. Shard. ② : Source Shard sends the state of the target account to Dest. Shard. ③ : Dest. Shard announces that the migration of the target account has been accomplished. ④ : Source Shard sends *new state* and associated pending TXs to Dest. Shard.

IV. Design of Account Migration Protocol

A. A Basic Workflow of Proposed Protocol

A basic workflow of the proposed account migration protocol is illustrated in Fig. 3, in which there are two types of roles in the proposed protocol, *i.e., Account Migration Trigger* and *Shard*.

- **Account Migration Trigger.** There should be a role that decides on account migration. We call it *Account (Acc.) migration trigger*, which is a trusted coordinator that runs the account-partition algorithm. It could be a trusted third party or a beacon committee. For example, in BrokerChain [15], to reduce the ratio of cross-shard TXs and balance the workload across shards, the so-called *reference committee* makes migration decisions by running an account-partition algorithm. Such the *reference committee* is an example of the trusted coordinator.
- **Shard.** Each blockchain shard maintains its local blockchain and stores a subset of all accounts in the whole system. In addition to packing the regular TXs that transfer tokens from one account to another, blockchain nodes in each shard should also pack the TXs related to account migration into blocks and achieve the intra-shard consensus. Since an account is not located in a fixed shard all the time, each shard needs to store the location of all accounts.

We then use Fig. 3 to demonstrate the basic workflow when migrating an account from its *source shard* to *destination (dest.) shard*. Once the decision to migrate an account is made by the Acc. migration trigger in step ① , it will be sent to *source shard* as a special *auxiliary TX* (defined in section IV-B). In step ② , when the nodes in *source shard* have reached a consensus on the auxiliary TX, they send the state of the account to *dest. shard*. When nodes in *dest. shard* agree to accept the target account, they add the state of the account to their local storage. Then in step ③ , *dest. shard* sends all other shards announcement TXs to notify that the target account is now located at *dest. shard*. Finally, each shard maps the target account to the *dest. shard*.

Furthermore, to keep fewer TXs related to the target account being suspended by the migration, *source shard* can still handle the TXs that *deposit* money to the target account (*payee TXs*) when it is being migrated across shards. When receiving the announcement from *dest. shard*, as shown in step ④ , nodes in the *source shard* begin to send the new state of the target account and the related pending TXs to *dest. shard*. The *dest. shard* will then modify the state of the target account according to the newly received state.

B. New Data Structures Defined in Our Protocol

In our protocol, the **state of an account** is represented by:

$$State = \{\texttt{Addr}, \texttt{Value}, \texttt{Nonce}, \texttt{Location}\},$$

where Addr is the address of an account, and Value represents the balance of the account. Nonce is an attribute of the account that denotes the number of TXs initiated by that account. Location tells where the account is located. For a specific shard S, when accepting a move-in account, it will modify the Location field in the account's state to '***S***'. When shard *S* is migrating the account to shard *D*, shard *S* modifies the Location field to '***to D***'. Once the migration is finished, shard *S* modifies the Location field to '***D***'. Even if the account is moved, shard *S* will not delete the state of that account, thus ensuring the traceability of the account in the blockchain.

In addition, in order to know which shard an account is located at, each shard must maintain all accounts' states locally. However, a shard only stores the Location field of an account in the state if the account is not managed by the shard. Take *other shards* in Fig. 3 as an example, they only store the Location of the target account under migration. Once receiving the announcement that the account has been migrated from its *source shard* to the *dest. shard*, other shards modify the Location field to '***Dest. Shard***'.

Auxiliary TXs. Auxiliary TXs are the transactions related to migrating an account. There are four types of auxiliary TXs in our protocol, *i.e.,* (i) $\texttt{TX}_{aux1}$ sent from *Acc. migration trigger*, (ii) $\texttt{TX}_{aux2}$ sent from *source shard*, (iii) $\texttt{TX}_{ann}$ sent from *dest. shard* to broadcast the latest location of the target account under migration, and (iv) $\texttt{TX}_{ns}$ sent from *source shard* to *dest. shard*.

(i) $\textbf{TX}_{aux1}$: When *Acc. migration trigger* makes a decision for account migration, a $\texttt{TX}_{aux1}$ is sent to the shard where the account is located. The structure of $\texttt{TX}_{aux1}$ is represented as follows:

$$\texttt{TX}_{aux1} = \{\texttt{Addr}, \texttt{Source}, \texttt{Dest.}, \tau\},$$

where field Addr is the address of the account to be migrated, Source indicates the shard that the account is currently located at, and field Dest. denotes the shard that the account is designated to. The field τ is the time limit for the migration operation. If the migration is not completed within time τ, the migration is considered to be failed.

(ii) $\textbf{TX}_{aux2}$: After reaching consensus on $\texttt{TX}_{aux1}$, the *source shard* will generate a $\texttt{TX}_{aux2}$ and send it to the *dest. shard*. The structure of $\texttt{TX}_{aux2}$ is defined as:

$$\texttt{TX}_{aux2} = \{\texttt{TX}_{aux1}, \texttt{State}, \texttt{H}, \texttt{MP}_{aux1}, \texttt{MP}_{state}\},$$

where State denotes the state of the account under migration and H refers to the height of the block on *source shard*'s chain that stores $\texttt{TX}_{aux1}$. $\texttt{MP}_{aux1}$ is the Merkle Proof to prove that $\texttt{TX}_{aux1}$ is stored in block H, and $\texttt{MP}_{state}$ proves the state of the account stored in the block.

(iii) $\textbf{TX}_{ann}$: Once nodes in *dest. shard* have reached a consensus on the migration and accept the move-in account, the *dest. shard* will broadcast $\texttt{TX}_{ann}$ messages to all other shards. The purpose of $\texttt{TX}_{ann}$ is to let all shards map the target account to *dest. shard*. The structure of $\texttt{TX}_{ann}$ is illustrated as follows:

$$\texttt{TX}_{ann} = \{\texttt{TX}_{aux2}, \texttt{State}, \texttt{H}, \texttt{MP}_{aux2}, \texttt{MP}_{state}\},$$

where State denotes the target account's state stored in *dest. shard* and H refers to the height of the block that stores $\texttt{TX}_{aux2}$. $\texttt{MP}_{aux2}$ and $\texttt{MP}_{state}$ are corresponding Merkle Proofs to prove the existence of $\texttt{TX}_{aux2}$ and State in block H, respectively.

(iv) $\textbf{TX}_{ns}$: Upon reaching consensus on the $\texttt{TX}_{ann}$, *source shard* sends $\texttt{TX}_{ns}$ to *dest. shard*. $\texttt{TX}_{ns}$ includes the new state of the account stored in *source shard*. The structure of $\texttt{TX}_{ns}$ is designed as follows:

$$\texttt{TX}_{ns} = \{\texttt{TX}_{ann}, \texttt{NewState}, \texttt{H}, \texttt{MP}_{ann}, \texttt{MP}_{newstate}\},$$

where NewState is the newest state of the target account stored in *source shard* and H refers to the height of the block that stores $\texttt{TX}_{ann}$. $\texttt{MP}_{ann}$ and $\texttt{MP}_{newstate}$ are corresponding Merkle Proofs of $\texttt{TX}_{ann}$ and NewState in block H, respectively.

Block. Fig. 4 shows the modified data structure of a block in our protocol. In addition to regular TXs, the block body also contains *auxiliary TXs*. Thus, we design a new *Merkle Patricia Trie* (MPT) [31] for indexing auxiliary TXs.

C. The Proposed Account Migration Protocol

The aim of our protocol is to prevent the TXs associated with the target account from being affected by the migration as much as possible. According to the *SOTA Lock* mechanism [18], no associated TXs can be processed at all, thereby prolonging the makespan of these TXs. Differently, our proposed protocol allows the execution of *payee TXs* even when the target account is being migrated. Fig. 5 shows the four stages of migrating an account in our protocol.

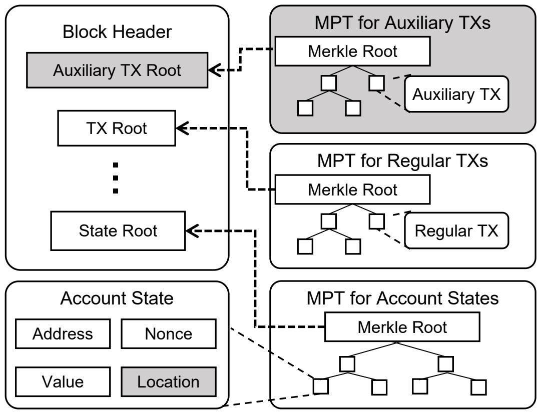


Fig. 4. Modified block's structure in the proposed protocol. Note that, the parts highlighted in dark shadows represent the components newly added.

Stage 1. A $\mathtt{TX}_{aux1}$, which aims to migrate an account to *dest. shard*, is sent to *source shard* and is waiting to be committed. The $\mathtt{TX}_{aux1}$ will be put into the *auxiliary pool* of *source shard*. *Auxiliary pool* is similar to *mempool*, but is used to store pending auxiliary TXs, while *mempool* stores pending regular TXs. After achieving consensus on $\mathtt{TX}_{aux1}$, *source shard* modifies the `Location` field to '***To Dest. Shard***' in the state of the target account. *Source shard* will not directly map the account to *dest. shard* until the migration is successful. Then, a $\mathtt{TX}_{aux2}$ is sent to *dest. shard*. Meanwhile, *source shard* will create a new *locking pool* to store pending *payer TXs* initiated by the target account. However, *payee TXs* can still be processed in *source shard* until the migration is completed.

Stage 2. Similar to $\mathtt{TX}_{aux1}$, $\mathtt{TX}_{aux2}$ will be waiting in the *auxiliary pool* of *dest. shard*. Before packing it into a block, *dest. shard* will check the validity of $\mathtt{TX}_{aux2}$ using the following three-step verification. Firstly, *dest. shard* checks if the time limit is exceeded. Secondly, *dest. shard* uses $\mathtt{MP}_{aux1}$ to check whether $\mathtt{TX}_{aux1}$ is stored in the block $\mathtt{H}$ of *source shard*'s blockchain. Finally, *dest. shard* verifies the state of the target account under migration by monitoring the `Location` field and then uses $\mathtt{MP}_{state}$ to verify its existence in the block $\mathtt{H}$. If all validations are correct, *dest. shard* will commit the $\mathtt{TX}_{aux2}$ and then modify the `Location` field in the state of the account to '***Dest. Shard***'. Afterward, *dest. shard* will send an announcement $\mathtt{TX}_{ann}$ to all shards notifying that the target account is now located at *dest. shard*, along with the Merkle Proof to demonstrate that the migration has been committed. From this moment on, *dest. shard* unlocks the account and begins to process any TXs related to the target account. Furthermore, *dest. shard* should record the account's balance *value*$_1$ at this time, which will be used in *Stage 4*. After broadcasting $\mathtt{TX}_{ann}$ to all shards, *dest. shard* can process any TXs related to the target account. Those TXs newly initiated by the target account will be routed to and processed by *dest. shard*.

Stage 3. After receiving the announcement from *dest. shard*, *source shard* will run a consensus on the announcement. Subsequently, *source shard* will modify the `Location` field of the account's state to '***Dest. Shard***', which means that

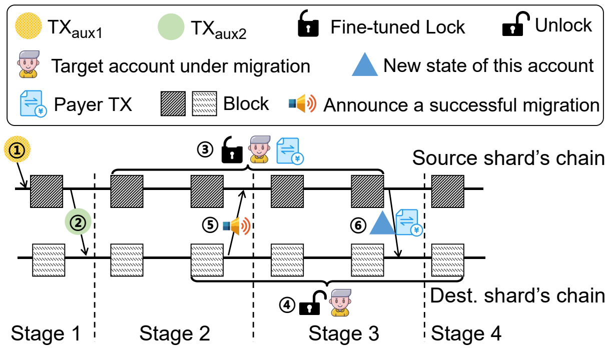


Fig. 5. A successful account migration while applying our protocol. ① : A migration request $\mathtt{TX}_{aux1}$ is sent to Source Shard. ② : Source Shard sends $\mathtt{TX}_{aux2}$ that includes state of the target account to Dest. Shard. ③ : Source Shard locks the target account and its associated payer TXs. ④ : Dest. Shard accepts and unlocks the target account. ⑤ : Dest. Shard announces to all other shards about this successful migration. ⑥ : Source Shard sends the new state of the target account together with its associated payer TXs.

source shard will no longer process any TXs related to the target account. Finally, *source shard* sends $\mathtt{TX}_{ns}$ that includes the newest state of the account to *dest. shard*, along with all *payer TXs* pending in the *locking pool*.

Stage 4. Once reaching consensus on the newest state of the target account from *source shard*, *dest. shard* firstly records the balance *value*$_2$ and calculates the change of target account's balance in *source shard* by $\Delta = value_2 - value_1$. *Dest. shard* will add Δ to the balance in the account's state stored in *dest. shard*. The pending TXs sent along with the newest state will be put into the *mempool* of *dest. shard*.

D. Handling Two Types of Migration Failures

During the account migration, our protocol may experience the following two types of failures. **Type 1:** The migration window τ expires in both source and destination shards. **Type 2:** The destination shard successfully verifies the account migration but the source shard does not perceive it due to the migration window τ expires.

Fig. 6 illustrates how the migration failure **Type 1** occurs. When the *dest. shard* fails to verify the target account via interpreting $\mathtt{TX}_{aux2}$, *dest. shard* will abort the target account and reply nothing to *source shard*. Moreover, if *source shard* has not received the announcement within the migration window τ, the migration is regarded as failed. Then, *dest. shard* will not pack $\mathtt{TX}_{aux2}$ into a block. Instead, *source shard* will initiate a query to *dest. shard* to check if the migration is

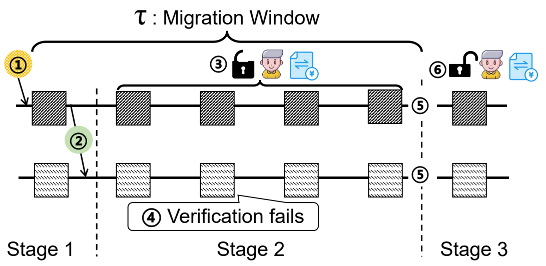


Fig. 6. Demonstration of **type 1** migration failure. ④ : Dest. Shard fails to verify the target account and replies nothing. ⑤ : The migration of the target account is found timing out through interpreting $\mathtt{TX}_{aux1}$ and $\mathtt{TX}_{aux2}$. ⑥ : Source Shard unlocks the target account and the associated payer TXs.

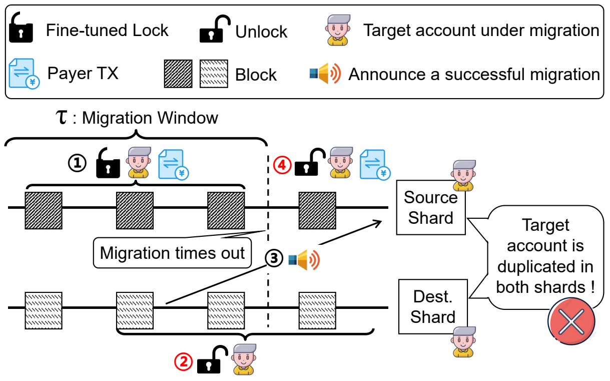

Fine-tuned Lock
Unlock
Target account under migration
Payer TX
Block
Announce a successful migration
τ : Migration Window
Source Shard
Dest. Shard
Migration times out
Target account is duplicated in both shards !
① ② ③ ④

Fig. 7. Demonstration of **type 2** failure, in which the target account is duplicated in both source and dest. shards.

successful. *Dest. shard* will reply with $\texttt{TX}_{ann}$ if the migration is successful. Otherwise, it will respond with a failure message along with proof showing not accepting the target account. When receiving a failure message, *source shard* will unlock the target account by changing the `Location` field in the account's state from '***To Dest. Shard***' back to '***Source Shard***'. Also, the pending TXs in the *locking pool* will be put back into the *mempool*.

Fig. 7 demonstrates **Type 2** failure, in which $\texttt{TX}_{ann}$ reaches *source shard* at a late time due to the propagation delay. As a result, the migration window τ expires. When τ is perceived expired, *source shard* should not directly unlock the target account. Otherwise, it will result in a wrong state in the source shard, because the target account will be duplicated in both shards. As a restoration strategy, *source shard* will initiate a new inquiry from the *dest. shard* to figure out whether the account migration is successful or not, until the correct result is perceived by the *source shard*.

E. Defending against Replay Attacks

We adopt the *relay transaction* mechanism [9] to handle cross-shard TXs (i.e., the TXs that the payer and payee locate at different shards). In the original *relay transaction* mechanism, when the payer's shard is handling a cross-shard TX (say `CTX`), nodes in the shard execute the deduction operation and then generate a relay TX. The relay TX will be sent to the payee's shard that manages the payee's account, with the Merkle Proof that `CTX` has been committed in the payer's shard. Payee's shard will execute the deposit operation once the relay TX is committed in the shard. However, in the system that enables account migration, if the payee's account is being migrated, a malicious node in the payer's shard can launch an attack by sending the relay TX to both the *source* and *dest.* shards. This type of attack is called ***replay attack***.

How a replay attack happens. We use Fig. 8 to illustrate this type of *replay attack*. During the period when *dest. shard* has sent an announcement but *source shard* has not received it, both of them can execute the deposit operation to the target account. If a malicious node from the payer's shard sends relay TXs to both shards, the deposit operation will be executed twice. Thus, this type of *replay attack* inflates the blockchain

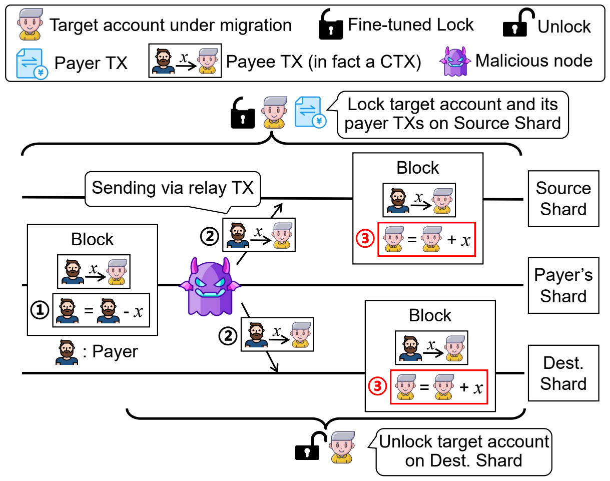


Fig. 8. Demonstration of Replay Attack. ① : A *payee* TX (actually a cross-shard TX) is packed in a block of Payer's Shard. ② : A malicious node in Payer's Shard sends two relay TXs (TX_{relay}) to both Source Shard and Dest. Shard. ③ : Both shards deposit a volume x of tokens to the target account.

system because an extra volume of tokens is generated by the malicious relay TXs.

How to deal with replay attacks. Such *replay attack* can be prevented by re-designing the structure of relay TX properly. We devise the structure as follows:

$$\texttt{TX}_{relay} = \{\texttt{CTX}, \texttt{State}_{rec}, \texttt{H}, \texttt{MP}_{ctx}, \texttt{MP}_{rec}\},$$

in which $\texttt{State}_{rec}$ and $\texttt{MP}_{rec}$ are the new attributes that are different from the original data structure of $\texttt{TX}_{relay}$. $\texttt{State}_{rec}$ is the payee's state stored in the payer's shard (only the `Location` field has a filled value), `H` denotes the height of the block that commits `CTX`, $\texttt{MP}_{ctx}$ and $\texttt{MP}_{rec}$ represent the Merkle Proofs that prove `CTX` and the state of the payee account in that block, respectively.

Note that, nodes in the payer's shard modify the `Location` of the payee to '***Dest. Shard***' when obtaining an announcement from *dest. shard*. Thus, if *source shard* receives the relay TX sent from payer's shard, the relay TX will be aborted since the `Location` field is not '***Source Shard***'.

V. Property Analysis of the Proposed Protocol

Since account migration needs to be recorded on the blockchain, our proposed protocol needs to guarantee the properties of *liveness* and *consistency* [32], [33]. The definitions for these two properties of account migration are given as follows.

Definition 1. *(Migration Liveness.) Every account migration will eventually be handled following either the successful procedure or the failure handling.*

Definition 2. *(Eventual Consistency of Migration.) For any account under migration, all shards will finally have the identical field of `Location` for the target account.*

Theorem 1. *As long as the source and the dest. shards are not corrupted by malicious nodes, the proposed Fine-tuned Lock mechanism achieves the liveness property (Definition 1).*

Proof. When *dest. shard* verifies the $\texttt{TX}_{aux2}$ sent from *source shard*, if the $\texttt{TX}_{aux2}$ is legal and has not timed out, as long as *dest. shard* is not corrupted by malicious nodes, it will notify all shards that the migration has been successful. On the other hand, if the $\texttt{TX}_{aux2}$ has timed out when *dest. shard* verifies it, *dest. shard* will not accept it. When *source shard* receives no announcement within the limited time or a failure message from *dest. shard*, if *source shard* is not corrupted by malicious nodes, it will consider the migration failed and unlock the target account. In summary, every account migration will be eventually handled following either the successful or failed handling. Thus, Theorem 1 concludes. □

Theorem 2. *As long as the source and the dest. shards are not corrupted by malicious nodes, the Fine-tuned Lock mechanism achieves the property of Eventual Consistency (Definition 2).*

Proof. For a $\texttt{TX}_{aux2}$, if not committed by *dest. shard* due to illegality or timeout reasons, the corresponding announcements will not be sent to any other shard. Even if a malicious node in *dest. shard* sends announcements to other shards, since there is no valid Merkle Proof to prove that $\texttt{TX}_{aux2}$ is committed, the announcement will be ignored. When the migration times out, *source shard* will modify the field `Location` of the target account back to '***Source Shard***', as long as the shard is not corrupted by malicious nodes. As a result, all shards still hold the view that the account is managed by *source shard.* On the other hand, if the $\texttt{TX}_{aux2}$ is committed by *dest. shard*, as long as the shard is not corrupted by malicious nodes, valid announcements will be sent to other shards. Finally, all shards map the account to *dest. shard.* In summary, all shards have the identical field of *`Location`* for the target account. □

Remark 1. *Theorems 1 and 2 imply that to guarantee the Liveness (Definition 1) and Eventual Consistency (Definition 2), neither source shard nor dest. shard should be corrupted by malicious nodes. This requires that the proportion of malicious nodes inside source shard and dest. shard cannot exceed 1/3, since our protocol adopts PBFT as the intra-shard consensus.*

Next, we calculate the probability that both *source shard* and *dest. shard* are secure.

Theorem 3. *Given a number $O \in \mathbb{N}^+$ of blockchain nodes, with $M \in \mathbb{N}^+$ being malicious nodes, each shard consisting of $n \in \mathbb{N}^+$ ($n < O$) nodes, and all shards adopting PBFT as their intra-shard consensus, the probability that both source and dest. shards are not corrupted by malicious nodes is:*

$$P = 1 - 2 \times \sum_{m=\lfloor n/3 \rfloor+1}^{min\{n,M\}} \frac{\binom{M}{m}\binom{O-M}{n-m}}{\binom{O}{n}} + \sum_{m_s=\lfloor n/3 \rfloor+1}^{min\{n,M\}} \sum_{m_d=\lfloor n/3 \rfloor+1}^{min\{n,M-m_s\}} \frac{\binom{n}{m_s}\binom{n}{m_d}\binom{O-2n}{M-m_s-m_d}}{\binom{O}{M}}. \quad (1)$$

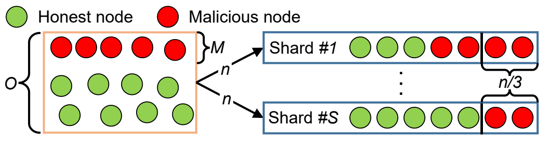


Fig. 9. Allocating a number O of blockchain nodes (M nodes are malicious) to S shards randomly and uniformly. Each shard has a number n of nodes.

Proof. We present our proof using Fig. 9. Firstly, the probability that the number m of malicious nodes in a shard exceeds $n/3$ can be calculated as:

$$P(m{>}\lfloor n/3 \rfloor) = \sum_{m=\lfloor n/3 \rfloor+1}^{min\{n,M\}} \frac{\binom{M}{m}\binom{O-M}{n-m}}{\binom{O}{n}}. \quad (2)$$

Then, denoting $m_s \in \mathbb{N}^+$ and $m_d \in \mathbb{N}^+$ as the number of malicious nodes in *source shard* and *dest. shard* respectively, the probability that both *source shard* and *dest. shard* are secure can be calculated as:

$$P = 1 - [2 \cdot P(m{>}\lfloor n/3 \rfloor) - P(m_s, m_d{>}\lfloor n/3 \rfloor)], \quad (3)$$

where $P(m_s, m_d{>}\lfloor n/3 \rfloor)$ is the probability that the proportions of malicious nodes in *source* and *dest.* shards both exceed 1/3.

According to [34], the process of assigning nodes to each shard is actually a sampling process without replacement. Therefore, when there are already n nodes in *source shard*, and m_s of them are malicious nodes, the nodes assigned to *dest. shard* can only be n nodes out of the remaining $O - n$ nodes, among which the malicious nodes can only be m_d nodes from the remaining $M - m_s$ malicious nodes. Therefore, the probability that there are m_s malicious nodes in *source shard* and m_d malicious nodes in *dest. shard* can be represented as:

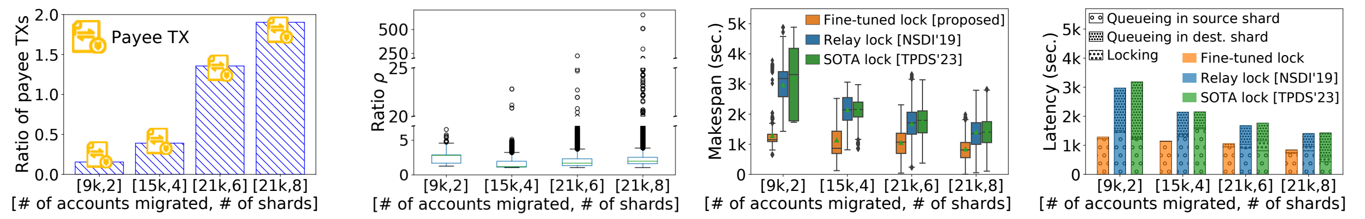
$$\begin{aligned} P(m_s, m_d) &= \frac{\binom{M}{m_s}\binom{O-M}{n-m_s}}{\binom{O}{n}} \cdot \frac{\binom{M-m_s}{m_d}\binom{O-n-(M-m_s)}{n-m_d}}{\binom{O-n}{n}} \\ &= \binom{n}{m_s}\binom{n}{m_d}\binom{O-2n}{M-m_s-m_d} \Big/ \binom{O}{M}. \end{aligned} \quad (4)$$

Thus, the probability that the proportion of malicious nodes in *source shard* and *dest. shard* both exceed 1/3 is:

$$\begin{aligned} P(m_s, m_d{>}\lfloor n/3 \rfloor) &= \sum_{m_s=\lfloor n/3 \rfloor+1}^{min\{n,M\}} \sum_{m_d=\lfloor n/3 \rfloor+1}^{min\{n,M-m_s\}} P(m_s, m_d) \\ &= \sum_{m_s=\lfloor n/3 \rfloor+1}^{min\{n,M\}} \sum_{m_d=\lfloor n/3 \rfloor+1}^{min\{n,M-m_s\}} \frac{\binom{n}{m_s}\binom{n}{m_d}\binom{O-2n}{M-m_s-m_d}}{\binom{O}{M}}. \end{aligned} \quad (5)$$

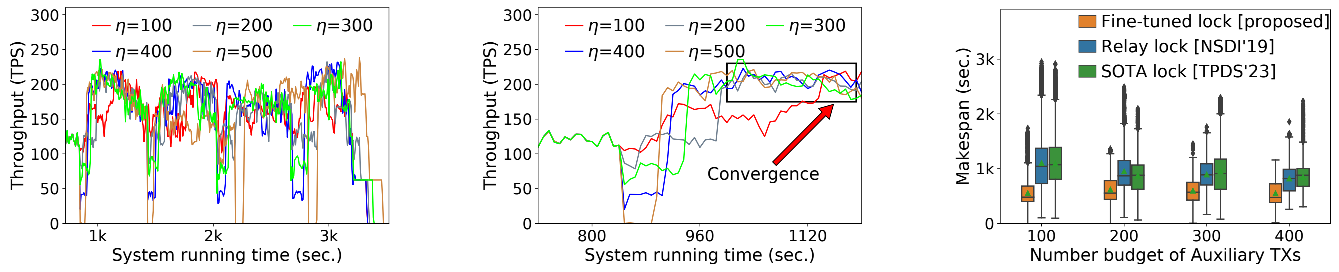
Finally, taking Eq. (5) in Eq. (3), Theorem 3 concludes. □

Remark 2. *It can be seen from Eq.* (1) *that, when the total number of nodes O and the number of malicious nodes M are fixed, setting the number n of nodes in each shard reasonably can ensure the security of source shard and dest. shard with a high probability. For example, when O=1000, M=150 and n=100, the probability P exceeds 99.9999%.*



(a) Ratio of locked *payee* TXs (SOTA) (b) Makespan affected by SOTA Lock (c) The makespan of *payee* TXs (d) Latency breakdown of *payee* TXs

Fig. 10. Multiple metrics of *payee TXs* versus various number of shards. Note that, in (a), the ratio of locked *payee* TXs is calculated by the # of locked *payee* TXs divided by the # of packed TXs in SOTA-Lock migration; in (b), ρ is a ratio between two terms (θ_1 and θ_2), θ_1 is the makespan of locked *payee TXs*, and θ_2 is the makespan of these TXs if they were not locked.



(a) TPS during the testing window (b) TPS around the first migration (c) The makespan of *payee* TXs

Fig. 11. System throughput and the makespan of *payee TXs* while setting different number budgets (denoted by η) for storing Auxiliary TXs in a block.

VI. Performance Evaluation

We implemented the proposed account migration protocol on top of an open-sourced blockchain testbed named *BlockEmulator* [35]. The fundamental functionalities of *BlockEmulator* include the following 4 points. i) It offers various customized consensus protocols. ii) It allows clients to import historical TXs into the transaction pool of the testbed. iii) It supports the built-in sharding mechanism. iv) It also provides multiple-metric measurement capabilities, such as recording transaction throughput, TX makespan, and other crucial metrics of a sharded blockchain. We then evaluate the proposed account migration protocol as well as two other locking mechanisms using *BlockEmulator*. The proposed protocol is implemented in GO with 5000+ lines of code.

A. Experiment Settings

Datasets. We collect the first 500 thousand TXs from Ethereum's historical data [31]. Every time 100,000 transactions are confirmed, a round of account migration is performed by invoking algorithm *CLPA* [17].

Transaction processing. Each shard block yielded by *BlockEmulator* can pack up to 500 TXs (including regular and auxiliary TXs). The number budget of auxiliary TXs in a block is set to 300 if not specified otherwise. In our experiments, a client is exploited to continuously feed sequential historical Ethereum TXs to the blockchain system at a specific rate. TXs will be injected into the shard where the payer of the TXs locates (*i.e.,* the payer's shard). Each shard node queues the arrived TXs according to the sequence of their arrival.

Baselines. In addition to the state-of-the-art lock mechanism (abbr. as *SOTA Lock*) proposed in *LB-Chain* [18], we also implement another lock mechanism called *Relay Lock*, inspired by the relay transaction mechanism proposed in *Monoxide* [9]. In *Relay Lock* mechanism, *payee TX* also needs to be locked when the payee account is under migration. However, this TX can be locked after the deduction operation is performed on the payer, if the payer of the TX is not under migration.

Deploying our protocol in Tencent Cloud. The number of blockchain shards is set to $2 \sim 8$ and each shard has 4 blockchain nodes. PBFT is adopted as the intra-shard consensus. We run our protocol in *BlockEmulator*, which runs in 32 virtual machine nodes rented from Tencent Cloud. Each virtual machine is deployed with a blockchain node, which is equipped with 16 CPU cores (Intel Xeon Cascade Lake 8255C) and 32 GB memory. The network bandwidth is set to 20 Mbps.

B. The Effect of Account Migration on Payee TXs

We first evaluate the effect of account migration on *payee TXs* under various # of migration accounts and the # of shards. The TX arrival rate ranges from $200 \sim 800$ transactions per second. In different configurations, the number of accounts required to migrate also varies based on the *CLPA* result. As observed in Fig. 10(a), during *SOTA-Lock* migration, the ratio of the number of locked *payee TXs* divided by the number of packed TXs increases with the number of shards. This observation implies that a larger number of shards induces more accounts that need to be migrated at each migration. Thus, the ratio of locked *payee TXs* also increases.

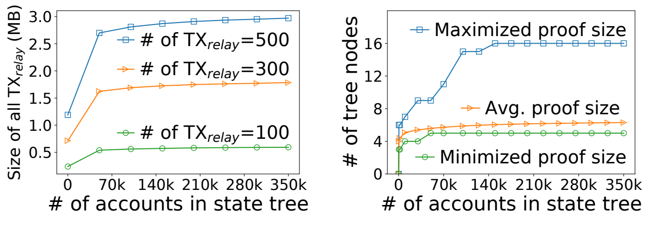
Fig. 10(b) demonstrates to what degree the makespan of *payee TXs* is prolonged when using *SOTA Lock* mechanism [18]. Here, ρ refers to the ratio of the makespan for locked *payee TXs* divided by the makespan of these TXs if they were not locked. It can be observed that regardless of the number of shards, the majority of TXs have a ratio ρ lower than 5. However, as the number of shards increases, the peak value of ρ also increases. When the shard number is 8, the peak value of ρ can exceed 600. The above observation shows that locking *payee TXs* has a significant impact on the TX makespan.

Fig. 10(c) shows the makespan of *payee TXs* using three lock mechanisms, respectively. The makespan of *payee TXs* decreases as the number of shards increases. Nonetheless, among the three mechanisms, the makespan of *payee TXs* in *Fine-tuned Lock* method remains the shortest. The makespan in other methods contains not only the period when the *payee TXs* are locked but also includes the queueing time in the *dest. shard* of target accounts under migration.

To better understand all latency terms in the entire makespan of *payee TXs*, Fig. 10(d) shows the breakdown of the makespan of *payee TXs*. We can observe that *payee TXs* are not locked in *Fine-tuned Lock* method, thus they are quickly processed in *source shard*. Furthermore, even in the remaining two methods, these TXs are locked for a relatively short duration, and the main factor contributing to the increased makespan is the queuing time in *dest. shard*. Therefore, if the mempool of *dest. shard* is filled with too many TXs, *payee TXs* will wait for a long time after being unlocked when applying *Relay Lock* and *SOTA Lock* methods.

C. The Number Budget of Auxiliary TXs in A Block

To examine the impact of the number budget (denoted by η) for storing auxiliary TXs in a block on the performance of the sharded blockchain, we vary η from 100 to 500. The number of shards and TX arrival rate are set to 4 and 200 TXs per second, respectively. Fig. 11(a) shows the real-time throughput (TPS) of the blockchain system. The TPS declines at each migration as the auxiliary TXs in the block compete for the available space with regular TXs. As observed, a higher value of η brings a lower real-time TPS during the account's migration. When $\eta = 500$, the TPS degrades to 0. Thereby, to prevent the system's TPS from significantly decreasing during account migration, it is suggested not to set an excessively high number budget η. On the other hand, the number budget η should not be set too low. Fig. 11(b) illustrates the zoomed-in TPS around the first migration. It is clear to see that a lower value of η requires a longer system running time to achieve the converged maximum system TPS. This is because a low η indicates the space budget in a block that can be used to store auxiliary TXs is small. Thus, the system needs more blocks to pack all the auxiliary TXs associated with the target account when η is set low. This means the system needs more running time to accomplish the migration of the target account. When η is set to 100, it takes 296 seconds to complete account migration. When η is set to 500, the system completes the account migration at the 920th second. Furthermore, it can be seen in Fig. 11(c) that the makespan of *payee TXs* does not vary significantly with changes of η. The *Fine-tuned Lock* mechanism still outperforms the other two mechanisms. Therefore, to strive for a balance between the system's TPS during account migration and the time spent on the account migration, it is suggested to set η to a moderate value. For example, η is set to 300 by default in our implementation.



(a) Data size of all $\mathtt{TX}_{relay}$ (b) Size of Merkle Proof

Fig. 12. Performance while varying the # of accounts in the state tree. Here, the size of *Merkle Proof* refers to the number of tree nodes in a given *proof*.

D. Data Size of TX_{relay} vs the # of Accounts

Finally, we study the data size of the modified $\mathtt{TX}_{relay}$, since the data size of all $\mathtt{TX}_{relay}$ determines the consumption of network bandwidth. Referring to section IV-E, the modified $\mathtt{TX}_{relay}$ includes two additional attributes, i.e., the payee's state $\mathtt{State}_{rec}$ and the corresponding Merkle Proof $\mathtt{MP}_{rec}$. Thus, we are curious about whether the total size of all $\mathtt{TX}_{relay}$ increases substantially or not while the account number grows.

By fixing the capacity of a block to 500 transactions, we carried out 3 groups of evaluation while increasing the number of accounts from 0 to 350 thousand. The zero account number represents that $\mathtt{TX}_{relay}$ is not modified. Fig. 12(a) shows that the size of modified $\mathtt{TX}_{relay}$ indeed increases following the growing number of accounts but quickly converges when the number of accounts exceeds 100 thousand. We use the results demonstrated in 12(b) to explain the observation from Fig. 12(a). When the number of accounts increases, the size of Merkle Proof (measured by the number of state tree nodes in the proof) also converges to 5 and 6 tree nodes in the categories of minimized proof size and the average proof size, respectively. Therefore, as the number of accounts increases, the data size of $\mathtt{TX}_{relay}$ does not grow significantly.

VII. Conclusion

To mitigate the impact of account migration on associated TXs, we study the efficient account migration for blockchain sharding. We first define the new block structure of the account's state. Then, we propose a *Fine-tuned Lock* mechanism to migrate an account while guaranteeing the *consistency* and *liveness* properties. Furthermore, we re-design the *relay* transaction to cope with the *replay attack* that may occur when migrating an account. We finally implemented our protocol on an open-sourced blockchain testbed named BlockEmulator and deployed it in the Tencent cloud. Experimental results demonstrate that the proposed protocol outperforms the other two lock mechanisms in terms of TX makespan.

Acknowledgement

This work was supported by the National Key R&D Program of China (No. 2022YFB2702304), NSFC (No. 62272496), Fundamental Research Funds for the Central Universities, (Sun Yat-sen University Grant No. 23lgbj019), and General Universities Key Field Project of Guangdong Province (No. 2023ZDZX1001).

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] 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.

[3] 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 41st International Conference on Distributed Computing Systems (ICDCS'21)*, 2021, pp. 629–639.

[4] B. David, B. Magri, C. Matt, J. B. Nielsen, and D. Tschudi, "Gearbox: Optimal-size shard committees by leveraging the safety-liveness dichotomy," in *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS'22)*, 2022, pp. 683–696.

[5] 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.

[6] 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 39th Symposium on Security and Privacy (SP'18)*, 2018, pp. 583–598.

[7] 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.

[8] 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.

[9] 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)*, 2019, pp. 95–112.

[10] 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 International Conference on Management of Data (SIGMOD'19)*, 2019, pp. 123–140.

[11] M. Li, Y. Lin, J. Zhang, and W. Wang, "Jenga: Orchestrating smart contracts in sharding-based blockchain for efficient processing," in *Proc. of IEEE 42nd International Conference on Distributed Computing Systems (ICDCS'22)*, 2022, pp. 133–143.

[12] Y. Liu, J. Liu, Q. Wu, H. Yu, Y. Hei, and Z. Zhou, "Sshc: A secure and scalable hybrid consensus protocol for sharding blockchains with a formal security framework," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, pp. 2070–2088, 2022.

[13] S. Li, M. Yu, C.-S. Yang, A. S. Avestimehr, S. Kannan, and P. Viswanath, "Polyshard: Coded sharding achieves linearly scaling efficiency and security simultaneously," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 249–261, 2020.

[14] M. Król, O. Ascigil, S. Rene, A. Sonnino, M. Al-Bassam, and E. Rivière, "Shard scheduler: object placement and migration in sharded account-based blockchains," in *Proc. of ACM Conference on Advances in Financial Technologies (AFT'21)*, 2021, pp. 43–56.

[15] H. Huang, X. Peng, J. Zhan, S. Zhang, Y. Lin, Z. Zheng, and S. Guo, "Brokerchain: A cross-shard blockchain protocol for account/balance-based state sharding," in *Proc. of IEEE 41st International Conference on Computer Communications (INFOCOM'22)*, 2022, pp. 1968–1977.

[16] C. Chen, Q. Ma, X. Chen, and J. Huang, "User distributions in shard-based blockchain network: Queueing modeling, game analysis, and protocol design," in *Proc. of 22nd International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing (MobiHoc'21)*, 2021, pp. 221–230.

[17] C. Li, H. Huang, Y. Zhao, X. Peng, R. Yang, Z. Zheng, and S. Guo, "Achieving scalability and load balance across blockchain shards for state sharding," in *Proc. of 41st International Symposium on Reliable Distributed Systems (SRDS'22)*. IEEE, 2022, pp. 284–294.

[18] M. Li, W. Wang, and J. Zhang, "Lb-chain: Load-balanced and low-latency blockchain sharding via account migration," *IEEE Transactions on Parallel and Distributed Systems*, 2023.

[19] E. Fynn, A. Bessani, and F. Pedone, "Smart contracts on the move," in *Proc. of 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'20)*, 2020, pp. 233–244.

[20] "Hyperledger cactus whitepaper." [Online]. Available: https://github.com/hyperledger/cactus/blob/main/whitepaper/whitepaper.md

[21] M. Sigwart, P. Frauenthaler, C. Spanring, M. Sober, and S. Schulte, "Decentralized cross-blockchain asset transfers," in *Proc. of 3rd International Conference on Blockchain Computing and Applications (BCCA'21)*. IEEE, 2021, pp. 34–41.

[22] Z. Hong, S. Guo, E. Zhou, J. Zhang, W. Chen, J. Liang, J. Zhang, and A. Zomaya, "Prophet: Conflict-free sharding blockchain via byzantine-tolerant deterministic ordering," in *Proc. of IEEE 42nd International Conference on Computer Communications (INFOCOM'23)*, 2023.

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

[24] 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.

[25] X. Qi, "S-store: A scalable data store towards permissioned blockchain sharding," in *Proc. of IEEE 41st International Conference on Computer Communications (INFOCOM'22)*, 2022, pp. 1978–1987.

[26] P. Zheng, Q. Xu, Z. Zheng, Z. Zhou, Y. Yan, and H. Zhang, "Meepo: Sharded consortium blockchain," in *Proc. of IEEE 37th International Conference on Data Engineering (ICDE'21)*, 2021, pp. 1847–1852.

[27] Y. Xu, T. Slaats, and B. Düdder, "Poster: Unanimous-majority-pushing blockchain sharding throughput to its limit," in *Proc. of ACM SIGSAC Conference on Computer and Communications Security (CCS'22)*, 2022, pp. 3495–3497.

[28] Z. Cai, J. Liang, W. Chen, Z. Hong, H.-N. Dai, J. Zhang, and Z. Zheng, "Benzene: Scaling blockchain with cooperation-based sharding," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 2, pp. 639–654, 2022.

[29] M. J. Amiri, D. Agrawal, and A. El Abbadi, "Sharper: Sharding permissioned blockchains over network clusters," in *Proc. of International Conference on Management of Data (SIGMOD'21)*, 2021, pp. 76–88.

[30] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)*, 1999, pp. 173–186.

[31] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, pp. 1–32, 2014.

[32] J. Garay, A. Kiayias, and N. Leonardos, "The bitcoin backbone protocol: Analysis and applications," in *Proc. of Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT'15)*. Springer, 2015, pp. 281–310.

[33] R. Pass, L. Seeman, and A. Shelat, "Analysis of the blockchain protocol in asynchronous networks," in *Proc. of Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT'17)*. Springer, 2017, pp. 643–673.

[34] A. Hafid, A. S. Hafid, and M. Samih, "A tractable probabilistic approach to analyze sybil attacks in sharding-based blockchain protocols," *IEEE Transactions on Emerging Topics in Computing*, vol. 11, no. 1, pp. 126–136, 2023.

[35] H. Huang, G. Ye, Q. Chen, Z. Yin, X. Luo, J. Lin, T. Li, Q. Yang, and Z. Zheng, "Blockemulator: An emulator enabling to test blockchain sharding protocols," *arXiv preprint arXiv:2311.03612*, 2023.