ShardTree: An Efficient Cross-Shard Protocol via Multi-party Virtual Payment Channel

***Abstract*—Sharding is a key approach to improving blockchain scalability for parallel transaction processing. Among various sharding strategies, state sharding splits the ledger across multiple shards, allowing each validator to process only a subset of transactions; however, maintaining consistency for cross-shard transactions remains a challenge. Most existing sharding systems lack scalability due to reliance on Merkle proofs, centralized intermediaries, or redundant ledger storage. In this paper, we propose ShardTree, a cross-shard protocol that enables high throughput and low confirmation latency for sharded blockchains without these limitations. ShardTree leverages a multi-party virtual payment channel (MPC) to efficiently manage cross-shard transaction processing in batches. We first design an algorithm to select validators to form an MPC with the maximum capacity over a payment channel (PC) path. Based on this, ShardTree constructs an MPC over a PC tree. We also design a rollback scheme to ensure the atomicity of cross-shard transactions. We theoretically prove that ShardTree guarantees security in the presence of Byzantine adversaries. Finally, we implement ShardTree and assess its performance using real-world Ethereum transactions. Evaluation results demonstrate that ShardTree efficiently processes cross-shard transactions and outperforms state-of-the-art protocols in terms of transaction throughput and confirmation latency.**

***Index Terms*—Blockchain, sharding, multi-party virtual payment channel.**

I. Introduction

Blockchains suffer from limited throughput due to the complexity of achieving global consensus. State sharding enhances scalability by partitioning the network into parallel-processing shards, each maintaining its own chain and reducing storage overhead. However, cross-shard transactions in a sharded blockchain introduce additional communication overhead due to several factors: 1) cross-shard message propagation, 2) redundant state verification by each shard, 3) atomicity enforcement to ensure all-or-nothing execution, and 4) synchronization overhead to maintain state consistency.

To enable cross-shard transaction processing, existing works have proposed various approaches [1–7], including account splitting, transaction relaying, two-phase commit approaches, etc. These methods rely on a single intermediary [1], require high-latency global proof collection [2–4], introduce cross-shard configuration [5–7], or incur additional storage overhead [2, 5]. To enable intra-shard transaction processing, each shard independently executes the intra-shard consensus protocol and maintains a sharded blockchain. Practical Byzantine Fault Tolerance (PBFT) protocol and its variants are widely used in existing sharding blockchains [1–3, 8–12], and industrial blockchains such as Hyperledger Fabric [13], Zilliqa [14], Tendermint [15], due to their resilience against Byzantine faults and support for immediate finality [16, 17].

In this paper, we propose a cross-shard protocol, ShardTree, that connects all shards using a multi-party virtual payment channel (MPC) [18]. An MPC is an off-chain channel that enables multi-sender multi-receiver transactions among multiple participants interconnected through payment channels (PC)s [19, 20]. In ShardTree, an MPC is formed by validators across shards, facilitating efficient cross-shard transactions. An MPC is tree-based if its underlying PC topology forms a tree, and path-based when that tree is a simple path. ShardTree offers the following **advantages** over state-of-the-art protocols: 1) ShardTree leverages the MPC formed by a selected set of validators who wish to participate in the MPC construction; 2) ShardTree offers better capacity to manage cross-shard transactions, compared to a single intermediary; 3) ShardTree does not split a user's account and store the split account state in different shards [1], nor does it create separate accounts for each incoming cross-shard transaction and wait for account aggregation after a certain period [6]; 4) ShardTree dynamically replenishes the MPC to sustainably support cross-shard transactions. Our main contributions are:

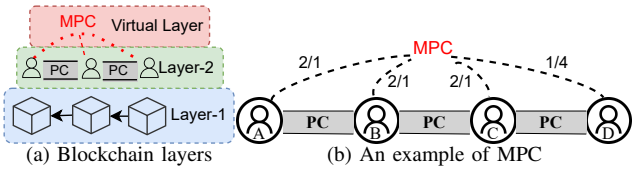
- We develop an algorithm that selects validators across shards to construct a path-based MPC with maximum capacity. Building on this, we propose ShardTree, which connects all shards via a tree-based MPC to efficiently handle batched cross-shard transactions.
- We design a rollback scheme for ShardTree to ensure the atomicity of the cross-shard transactions.
- We provide a security analysis demonstrating that ShardTree is resilient to up to $\frac{1}{3}$ Byzantine validators.
- Evaluation results show that ShardTree achieves low MPC construction latency and outperforms state-of-the-art cross-shard protocols in terms of transaction throughput and confirmation latency.

Organization. Section II overviews the background. Section III provides the system and threat model. Section IV describes the design of ShardTree. Section V illustrates the MPC validator selection and construction process. Section VI describes the cross-shard transaction processing. Section VII shows the security analysis. Section VIII provides evaluation results. Section IX concludes this paper.

II. Background

A. Payment Channel

A payment channel (PC) is an off-chain solution that enables two parties to exchange multiple off-chain payments, settling



(a) Blockchain layers (b) An example of MPC

Fig. 1: (a) Three layers of a blockchain system. (b) An example of an MPC: four parties share a channel with an initial total capacity of 7. Each party holds an initial and updated balance, denoted as initial/updated. A, B, and C each transfer 1 to D. The updated balances are: A, B, and C with 2/1, and D with 1/4. The total capacity remains the same before and after the transaction.

only the final state on-chain. The lifetime of a PC consists of three stages: open, update, and close (peacefully or dispute) [19, 20]. To open a PC, two parties exchange signatures on their initial balance and then lock their respective funds in the PC via a smart contract [19]. Then, they can freely exchange encrypted messages to update the state of the PC. Finally, to close the PC, a final state agreed by both parties is recorded on-chain, after which the correct funds are unlocked to the two participants. Either party is allowed to initiate a dispute, and the counter-party can respond within a challenge time window [19].

B. Multi-party Virtual Payment Channel

As shown in Fig. 1(a), the bottom layer (Layer-1) of the blockchain system is a decentralized ledger that records transactions immutably in blocks. The second layer (commonly referred to as Layer-2) is built over Layer-1 by executing PC contracts deployed on-chain. The "virtual" layer hosts virtual PCs, which are established through executing specialized virtual channel smart contracts. The open and optimistic close stages of these virtual channels are recorded on the PC, making both phases effectively off-chain [18, 21–23]. The disputes can be resolved directly on-chain, regardless of the number of parties involved [18]. This paper focuses on the multi-party virtual payment channel (MPC) constructed by executing the MPC smart contract over a tree of PCs, which significantly expands the applicability of virtual PCs [18, 21]. An example of a four-party MPC is illustrated in Fig. 1(b). The four parties, A, B, C, and D, are connected via a path of PCs. The four PCs are the *subchannels* of the MPC. All parties of the MPC can update the balances of parties off-chain.

III. System Model and Threat Model

In this section, we introduce the system and threat models.

A. System Model

A total of N validators are responsible for processing transactions. The validators in the whole blockchain network are divided into a set M *regular* shards for processing transactions and an *identity* shard for maintaining validator identities. Each identity shard S_i containing n validators, where $i \in [1, M]$. Each validator holds an Elliptic Curve Digital Signature Algorithm (ECDSA) public-private key pair for signing transactions [24]. An identity uniquely represents a validator using its public key, IP address, and a valid proof of work solution to the identity shard, as commonly adopted in prior works [1–3, 12]. For **intra-shard** processing, each regular shard operates as a committee responsible for batching transactions, achieving

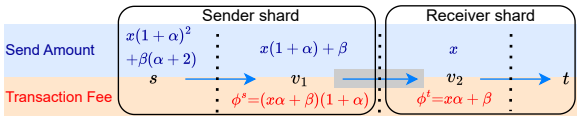


Fig. 2: Illustration of the fee model: a transaction of amount x is forwarded from sender s to receiver t through two MPC validators, v_1 and v_2, where s and v_1 reside in one shard, and t and v_2 reside in another.

intra-shard consensus, and appending them to its ledger. There is a randomly elected *consensus leader* for each shard that drives the process of PBFT consensus [25, 26]. For **cross-shard** processing, each shard must have at least one validator participating in the MPC to handle cross-shard transactions on its behalf. Let $\mathcal{V}_i$ denote the remaining validators of S_i (excluding the MPC validators of this shard).

Each cross-shard transaction is decomposed into two intra-shard transactions and an MPC transaction; thus, only two MPC validators participate in forwarding funds. Therefore, unlike traditional Hashed Timelock Contract-based routing approaches [27–30], where the number of intermediaries varies unpredictably—making fee pre-computation infeasible—the system model enforces a 3-hop path for cross-shard transactions. It involves exactly two intermediaries: one MPC validator from the *sender shard* and one from the *receiver shard*. The sender shard is where the sender's account resides; the receiver shard is where the receiver's account resides. We propose a fee model that incentivizes MPC validators, similar to existing PC implementations [20, 30–32]. The transaction fee consists of a proportional component α and a base fee β, as in prior work [31, 32]. The detailed fee breakdown for the fixed path length is illustrated in Fig. 2. Once the transaction amount x is set, each intermediary fee can be computed accordingly. We define the fee set $\phi = \{\phi^s, \phi^t\}$, where ϕ^s and ϕ^t are the fees for the MPC validators in the sender and receiver shard, respectively. This fee model ensures that each MPC validator is compensated for participating in processing cross-shard transactions, while bounding the total transaction cost.

B. Threat Model

Following widely adopted assumptions for PBFT consensus in existing sharding and security works [2, 3, 5, 6, 8–10, 25, 26], we consider a probabilistic polynomial-time Byzantine adversary that can corrupt $f < \frac{n}{3}$ of the validators at any given time, where f denotes the number of malicious validators in each shard. Malicious validators can collude with each other, such as sending incorrect messages or being unresponsive. Malicious adversaries are also slowly adaptive, which takes time to corrupt validators [3–5, 12]. The set of malicious and honest validators in a shard remains the same during each epoch, which can change only between epochs. Leaders can be malicious and refuse to forward transactions. We adopt a signature scheme $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$ as ECDSA [24], which is existentially unforgeable under chosen-message attacks.

IV. Design of ShardTree

In this section, we introduce the design of ShardTree. ShardTree proceeds in fixed time intervals called *epochs*. As shown in Fig. 3, each epoch consists of two phases: reconfiguration and consensus, similar to existing works [1, 3,

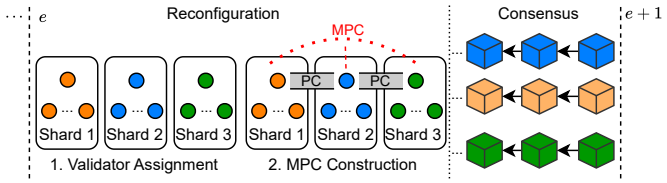


Fig. 3: Overview of ShardTree, where shard membership is visually distinguished by color. ShardTree proceeds in repeated *epochs*, with the previous epoch denoted by e and the current one by $e+1$. Each epoch comprises a reconfiguration phase followed by a consensus phase.

8, 12]. The reconfiguration phase primarily involves assigning validators to shards and constructing MPCs for use in the subsequent consensus phase. The consensus phase involves multiple rounds of PBFT to achieve consensus.

Reconfiguration: During the reconfiguration phase, validators partition all accounts in the current blockchain network into different shards using the Metis-based method, as commonly adopted by existing sharding works [1, 6, 33, 34]. Similar to existing works [4, 35, 36], we also use a verifiable random function (VRF) to securely select a random validator as the *epoch leader* l_e. Validators are assigned to shards using the Rand-Hound algorithm in a decentralized manner [3]. Each shard has a set of candidate validators for MPC construction. ShardTree then selects a subgroup from the candidate validators to construct an MPC as detailed in Section V.

Consensus: During the consensus phase, transactions are routed to the shard where the sender's account is located [6, 8, 9, 37]. Intra-shard transactions are processed entirely by the designated validators within a single shard using PBFT consensus. Cross-shard transactions are decomposed into two intra-shard transactions, processed within their respective shards, and one MPC transaction. The converted intra-shard transactions are processed identically to regular intra-shard transactions. The details of handling MPC update is detailed in Section V. For the consensus phase, we will focus on cross-shard transaction processing and the atomicity guarantee for transactions in Section VI.

V. Multi-party Virtual Payment Channel Construction

In this section, we first introduce the notations and constraints for MPC construction. We then formally define the underlying tree formation problem to maximize capacity for MPC. Next, we present the underlying path and tree formation algorithms in Algorithms 1 and 2, respectively. Finally, we show the details of the MPC operations in Algorithms 3–5.

A. Notations and Constraints

Notations: There exists a set of candidate MPC validators $\mathcal{M}$, desiring to participate in constructing the MPC. Each validator $v_i \in \mathcal{M}$ declares a budget b_i to lock in their PCs for the MPC construction. Let $\mathcal{B}$ denote the set of all budgets and $\mathcal{B}_i$ denote the set of budgets of validators in shard S_i, where $\mathcal{B}=\bigcup_{i=1}^{M}\mathcal{B}_i$. Let B denote the total budget, i.e., $B=\sum_{v_i\in\mathcal{M}} b_i$. Let $\tilde{\mathcal{B}}=\bigcup_{i=1}^{M}\max(\mathcal{B}_i)$ denote the budget set containing the maximum budget from each respective $\mathcal{B}_i$. Let $\hat{\mathcal{M}} \subseteq \mathcal{M}$ denote the selected subset of validators who

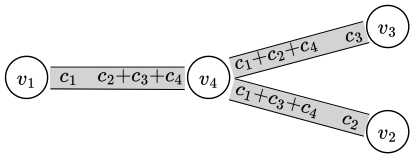


Fig. 4: Illustration of Constraint (1): In a PC tree, v_1, v_2, and v_3, only need to lock their own capacity share in their respective PC. However, v_4 must lock $c_1+c_2+c_4$ to the PC with v_3, because v_1, v_2 and v_4 are in the same connected component, assuming this PC is removed.

ultimately construct the MPC. Let $\hat{B}$ denote the total budget of validators in $\hat{\mathcal{M}}$. To construct an MPC, validators in $\hat{\mathcal{M}}$ are connected via PCs to form a tree $\hat{\mathcal{P}}$. Let $\mathcal{N}_i$ denote the set of neighbors of v_i on $\hat{\mathcal{P}}$. Let d_i be the degree of v_i on $\hat{\mathcal{P}}$. If $d_i>1$, v_i is an internal validator (IV); otherwise, it is an end validator (EV). Let c denote the capacity of the MPC and c_i denote the capacity share allocated to validator v_i in $\hat{\mathcal{M}}$, where $c=\sum_{v_i\in\hat{\mathcal{M}}} c_i$. Each validator v_i needs to lock at least c_i to its PCs, referred to as v_i's *investment*. Let $\hat{\mathcal{C}} = \{c_i | v_i \in \hat{\mathcal{M}}\}$ denote the set of capacity shares of validators in $\hat{\mathcal{M}}$.

Constraints: We have the following constraints for the validators in $\hat{\mathcal{M}}$:

(1) According to the MPC design [18, 21], each IV must lock enough balance on each of its PCs to match the total investment of all the validators in the same connected component, assuming this PC is removed. In addition, it must have enough budget to cover the total locked balance across all its PCs. Therefore, we have $c_i+(d_i-1)c \leqslant b_i$. An example is shown in Fig. 4.

(2) For fairness, each participating validator v_i has a capacity share proportional to its budget, i.e., $c_i=\frac{b_i}{B}c$.

(3) Each shard S_i must have at least one validator to participate the MPC construction to ensure all the shards are connected, where $S_i \cap \hat{\mathcal{M}} \neq \emptyset$.

By constraints (1) and (2), we obtain

$$c \leqslant \left(\frac{d_i-1}{b_i}+\frac{1}{B}\right)^{-1},\ \forall v_i \in \mathcal{M} \tag{1}$$

We define the right-hand side as the *effective budget* of v_i. If v_i is an EV, the constraint becomes $c \leqslant B$, which holds trivially by definition. When the underlying topology is a path, the constraint for any IV is $c \leqslant \frac{b_i B}{b_i+B}$, Thus, for a path topology, the maximum c is determined solely by the bottleneck effective budget of the IVs.

Validator	v_1	v_2	v_3	v_4
Budget	$b_1=2$	$b_2=4$	$b_3=4$	$b_4=2$
Eq. (1)	$c \leqslant 12$	$c \leqslant 3$	$c \leqslant 3$	$c \leqslant 12$

Table I: Illustration of the constraints when constructing a path-based MPC with four validators (v_1 and v_4 as EVs, v_2 and v_3 as IVs).

We use Table I to illustrate the constraints with a simple example of four validators constructing a path-based MPC. The objective is to find the maximum c this group can support. Given $c \leqslant 3$, we can get that the maximum capacity is 3. The resulting capacity shares of the four validators can be easily derived: $c_1=c_4=0.5$ and $c_2=c_3=1$.

B. Problem Formulation

When multiple validators desire to construct the MPC, maximizing its capacity becomes more challenging, as including all participants does not necessarily lead to the highest capacity. Specifically, it requires ensuring that each shard is represented by at least one validator and that the selected validator set—drawn from a flexible pool—maximizes the overall capacity while satisfying these constraints. In this paper, we focus on the following underlying MPC tree formation problem in ShardTree: given $\mathcal{M}$ and $\mathcal{B}$, we aim to design an strategy to form an underlying tree that constructs an MPC with high c across M shards.

C. Forming Optimal Underlying Path for MPC

We begin with the case where the underlying tree is a path. Before introducing the optimal algorithm for the underlying path formation, we first show a minimal example of three validators and show in Lemma 1 that assigning the two validators with the smallest budget as EVs maximizes capacity. We then extend to longer PC paths by enlarging the IV set and show in Lemma 2 that the maximum capacity is independent of the IV order, and in Lemma 3 that it remains optimal among all PC paths formed from the same validator set.

Lemma 1. *Validators v_1, v_2, and v_3 with budgets $b_1 \leqslant b_2 \leqslant b_3$ can construct a path-based MPC with the maximum capacity $\frac{b_3 B}{b_3+B}$ when v_1 and v_2 serve as EVs, where $B=\sum_{i=1}^{3} b_i$.*

Proof. There are three possible PC path topologies, corresponding to each validator v_1, v_2, and v_3 serving as the IV. Let c', c'' and c''' denote the maximum capacity when v_1, v_2, and v_3 serve as the IV, respectively. Using Eq. (1), we obtain the bottleneck effective budget: $c \leqslant \frac{b_1 B}{b_1+B}$. This gives us $c'=\frac{b_1 B}{b_1+B}$. Similarly, when v_2 and v_3 is the IV, we obtain $c''=\frac{b_2 B}{b_2+B}$ and $c'''=\frac{b_3 B}{b_3+B}$, respectively. As $\frac{b_3 B}{b_3+B} \geqslant \frac{b_2 B}{b_2+B} \geqslant \frac{b_1 B}{b_1+B}$, we conclude that $c''' \geqslant c'' \geqslant c'$. This completes the proof. □

Lemma 2. *$\tilde{n}$ validators with budgets $b_1 \leqslant b_2 \leqslant \ldots \leqslant b_{\tilde{n}}$ construct a path-based MPC, where $\tilde{n}>3$. Designate v_1 and v_2 as EVs and $U=\{v_3, v_4, \ldots, v_{\tilde{n}}\}$ as IVs, the following holds:*

(a) The maximum capacity is $\frac{b_3 B}{b_3+B}$.

(b) The maximum capacity is independent of the order of IVs.

Proof. Using Eq. (1), the constraint is given by: $c \leqslant \frac{b_i B}{b_i+B}$, where $B=\sum_{i=1}^{\tilde{n}} b_i, \forall v_i \in U$. Since the constraint for EVs always holds and the following inequality holds among IVs: $\frac{b_3 B}{b_3+B} \leqslant \frac{b_4 B}{b_4+B} \leqslant \ldots \leqslant \frac{b_{\tilde{n}} B}{b_{\tilde{n}}+B}$, this gives a bottleneck effective budget $\frac{b_3 B}{b_3+B}$. Thus, the maximum capacity is $\frac{b_3 B}{b_3+B}$, which proves (a). Since all IVs have a degree of two, reordering IVs along the path does not affect the bottleneck effective budget, which proves (b). This completes the proof. □

Lemma 3. *The maximum capacity $\frac{b_3 B}{b_3+B}$ from Lemma 2 remains optimal among all PC paths formed with the same validator set.*

Proof. Lemma 2 proves that the maximum capacity is independent of the order of IVs. Thus, we only need to consider topologies where EVs v_1, v_2, or both swap roles with IVs. We prove the lemma by contradiction, showing that the resulting maximum capacity is less than $\frac{b_3 B}{b_3+B}$. There are three contradictory cases: 1) v_1 becomes an IV, v_1 satisfies $c \leqslant \frac{b_1 B}{b_1+B}$. Since $\frac{b_1 B}{b_1+B} \leqslant \frac{b_i B}{b_i+B}, \forall v_i \in U$, we have the maximum capacity $\frac{b_1 B}{b_1+B} \leqslant \frac{b_3 B}{b_3+B}$; 2) v_2 becomes an IV, v_2 satisfies $c \leqslant \frac{b_2 B}{b_2+B}$, the maximum capacity is: $\frac{b_2 B}{b_2+B} \leqslant \frac{b_3 B}{b_3+B}$; and 3) both v_1 and v_2 become IVs. This reduces to *Case 1*. Hence, any such role swap leads to a lower capacity, completing the proof. □

Each shard may have multiple candidates, and adding more to the MPC construction does not necessarily increase capacity. Lemma 3 does not account for the constraint that each shard must have at least one validator in the MPC construction. Intuitively, validators with very small budgets should be excluded, as they may lower the bottleneck effective budget. Theorem 1 formalizes this insight under this constraint.

Fact 1. *$\frac{bB}{b+B} > \frac{b^- B^+}{b^- + B^+}$, where $B^+ > B > b > b^-$.*

Theorem 1. *The optimal capacity $\frac{\hat{b}\hat{B}}{\hat{b}+\hat{B}}$ is achieved when the MPC is constructed over a PC path, such that two validators with the smallest two budgets in $\tilde{\mathcal{B}}$ are designated as EVs, $\left\{v_j \in \mathcal{M} \mid b_j \geqslant \hat{b}\right\}$ designated as IVs, where $\hat{b}$ is the minimum budget in the rest of $\tilde{\mathcal{B}}$, and $\hat{B}$ is the budget summation of all validators on this PC path.*

Proof. We prove this theorem from two perspectives: IV selection and EV selection. First, we show by contradiction that including validators with budget below $\hat{b}$ or removing IVs with budget above $\hat{b}$ reduces capacity. **1)** Suppose a validator with b_j is included ($b_j < \hat{b}$). Let B' denote the total budget of all validators in $\hat{\mathcal{M}}$ after including the validator, where $B'=\hat{B}+b_j$. Thus, the maximum capacity after including the validator is $\frac{b_j B'}{b_j+B'}$, where the original maximum capacity is $\frac{\hat{b}\hat{B}}{\hat{b}+\hat{B}}$. Given Fact 1, we have $\frac{b_j B'}{b_j+B'} < \frac{\hat{b}\hat{B}}{\hat{b}+\hat{B}}$; **2)** Let B'' denote the budget sum after a validator is removed, where $B'' < \hat{B}$. For all the IVs, they satisfy $c \leqslant \frac{bB''}{b+B''}$. Lemma 2 proves that effective budget increases with increasing budget sum. Thus, $\frac{\hat{b}B''}{\hat{b}+B''} < \frac{\hat{b}\hat{B}}{\hat{b}+\hat{B}}$. Second, we prove that the two validators with the smallest budgets have to be designated as EVs to reach the maximum capacity when the underlying topology is a path in Lemma 3. This completes the proof. □

Based on Theorem 1, we design Algorithm 1 to form the underlying path for MPC. The epoch leader l_e shall run Algorithm 1 in the reconfiguration phase to initiate the MPC selection. Given a set of candidate validators $\mathcal{M}$, l_e determines the $\hat{b}$ as in Theorem 1 (Line 2). It then adds those validators with $b_i \geqslant \hat{b}$ to the selected validator set $\hat{\mathcal{M}}$, updating the total budget sum $\hat{B}$ accordingly (Lines 3 to 4). Then l_e chooses two validators with the smallest budgets from $\hat{\mathcal{M}}$ to serve as EVs on the PC path, and the rest validators in $\hat{\mathcal{M}}$ as IVs (Line 5). Note that the order of the IVs in the path $\hat{\mathcal{P}}$ does not affect the optimality of the capacity, as proven in Lemma 2. l_e then computes c based on Theorem 1 (Line 6) and $\hat{\mathcal{C}}$ for all validators (Line 7). $\hat{\mathcal{M}}$ execute the PC smart contract to

Algorithm 1: Underlying Path Formation for MPC

Input: candidate validator set $\mathcal{M}$, candidate budget set $\mathcal{B}$ and maximum budget set $\tilde{\mathcal{B}}$.
Output: capacity c, capacity share set $\hat{\mathcal{C}}$, selected validator set $\hat{\mathcal{M}}$, and PC path $\hat{\mathcal{P}}$.

1 $c \leftarrow 0$, $\hat{\mathcal{M}} \leftarrow \emptyset$, $\hat{\mathcal{C}} \leftarrow \emptyset$, $\hat{\mathcal{P}} \leftarrow \emptyset$, $\hat{B} \leftarrow 0$;
2 Obtain $\hat{b}$ and two EVs using $\tilde{\mathcal{B}}$ as in Theorem 1;
3 **foreach** $v_i \in \mathcal{M}$ with $b_i \in \mathcal{B}$ and $b_i \geqslant \hat{b}$ **do**
4 $\quad \hat{B} \leftarrow \hat{B}+b_i$, $\hat{\mathcal{M}} \leftarrow \hat{\mathcal{M}} \cup \{v_i\}$, Update IVs;
5 Update $\hat{\mathcal{P}}$ with obtained EVs and IVs as in Theorem 1;
6 $c \leftarrow \frac{\hat{b}\hat{B}}{\hat{b}+\hat{B}}$;
7 **foreach** $v_i \in \hat{\mathcal{M}}$ **do** $\hat{\mathcal{C}} \leftarrow \hat{\mathcal{C}} \cup \{\frac{b_i}{\hat{B}}c\}$;
8 **return** c, $\hat{\mathcal{C}}$, $\hat{\mathcal{M}}$, and $\hat{\mathcal{P}}$

Algorithm 2: Underlying Tree Formation for MPC

Input: candidate validator set $\mathcal{M}$, candidate budget set and $\mathcal{B}$ and maximum budget set $\tilde{\mathcal{B}}$
Output: capacity c, capacity share set $\hat{\mathcal{C}}$, selected validator set $\hat{\mathcal{M}}$, and PC tree $\hat{\mathcal{P}}$

1 Run `Algo1` ($\mathcal{M}$, $\mathcal{B}$, $\tilde{\mathcal{B}}$) to obtain $\hat{\mathcal{M}}$, $\hat{\mathcal{P}}$, $\hat{\mathcal{C}}$ and c;
2 **repeat**
3 $\quad$ Increase current c by add, remove or reorganize;
4 $\quad$ Update $\hat{\mathcal{C}}$, $\hat{\mathcal{M}}$, and $\hat{\mathcal{P}}$, accordingly;
5 **until** *no further c improvement*;
6 **return** c, $\hat{\mathcal{C}}$, $\hat{\mathcal{M}}$ and $\hat{\mathcal{P}}$

establish PCs based on $\hat{\mathcal{P}}$, forming the underlying path for the MPC [19, 20]. Algorithm 1 returns c, $\hat{\mathcal{C}}$, $\hat{\mathcal{M}}$, and $\hat{\mathcal{P}}$.

D. Forming Underlying Tree for MPC

We consider the special case where the underlying topology is a path in Section V-C. Unlike paths, the positions of IVs matter in a tree. Therefore, designing an optimal underlying tree formation algorithm is significantly challenging, and we leave a deeper investigation as future work. In this paper, we propose Algorithm 2 that constructs a tree based on the path formed in Algorithm 1.

Before introducing Algorithm 2, we first define three types of operations on IVs that might improve the capacity: 1) add: connect a validator with maximum budget from $\mathcal{M}\backslash\hat{\mathcal{M}}$ to the validator with the highest effective budget, subject to the constraint that doing so does not introduce a smaller bottleneck effective budget; 2) remove: remove the validator with the minimum budget, among those who are not the sole validator in their respective shards; and 3) reorganize: move the validator with the minimum budget and connect it to the validator with the highest effective budget, provided this does not introduce a smaller bottleneck effective budget.

In Algorithm 2, we first obtain initial $\hat{\mathcal{M}}$, $\hat{\mathcal{C}}$, $\hat{\mathcal{P}}$ and c by running Algorithm 1 (Line 1). Next, we repeatedly apply the operation—among the three types—that yields the largest increase in c, and update $\hat{\mathcal{M}}$, $\hat{\mathcal{C}}$ and $\hat{\mathcal{P}}$, accordingly (Lines 3 to 4). Iterations stop when no bigger c can be obtained (Line 5). $\hat{\mathcal{M}}$ execute the PC smart contract to set up PCs as specified by $\hat{\mathcal{P}}$, thereby forming the underlying path for the MPC [19, 20]. Algorithm 2 returns the updated c and corresponding sets $\hat{\mathcal{M}}$, $\hat{\mathcal{C}}$ and $\hat{\mathcal{P}}$ for constructing MPC.

Algorithm 3: MPC Construction

Input: selected validator set $\hat{\mathcal{M}}$ and start round τ_0.

1 l_e broadcasts message $(\texttt{construct}, \sigma_e)$;
2 **foreach** $v_i \in \hat{\mathcal{M}}$ **do**
$\quad$ // Upon receiving `construct` in τ_0
3 $\quad$ **foreach** $v_j \in \mathcal{N}_i$ **do**
4 $\quad\quad$ **if** $b_i \leqslant b_j$ **then**
5 $\quad\quad\quad$ v_i sends $(\texttt{open}, \sigma_i)$ to v_j in τ_0;
6 $\quad\quad$ **else**
$\quad\quad\quad$ // Upon receiving `open` in τ_0+1
7 $\quad\quad\quad$ v_i sends $(\texttt{ack}, \sigma_i)$ to v_j in τ_0+1;
8 $\quad\quad$ **if** v_j receives $(\texttt{ack}, \sigma_i)$ in τ_0+2 **then**
9 $\quad\quad\quad$ v_j sends $(\texttt{opened}, \sigma_j)$ to $\hat{\mathcal{M}}\backslash v_j$;
10 $\quad\quad$ **else** v_j terminates;
11 $\quad$ **foreach** $v_j \in \hat{\mathcal{M}}\backslash v_i$ **do**
12 $\quad\quad$ **if** v_j misses any `opened` in τ_0+3 **then**
13 $\quad\quad\quad$ v_j sends $(\texttt{rej}, \sigma_i)$ to $\hat{\mathcal{M}}\backslash v_j$;
14 $\quad\quad\quad$ v_j terminates;
15 $\quad\quad$ **if** v_j receives any `rej` in τ_0+4 **then**
16 $\quad\quad\quad$ v_j terminates;
17 $\quad\quad$ **else** v_j sends $(\texttt{created}, \sigma_j)$ to l_e;

Algorithm 4: MPC Update

Input: selected validator set $\hat{\mathcal{M}}$, initial state version number w and start round τ_0.

1 **foreach** $v_i \in \hat{\mathcal{M}}$ receiving `update` upon τ_0 **do**
2 $\quad$ v_i sends $(\texttt{update}, w, \sigma_i)$ to $\hat{\mathcal{M}}\backslash v_i$;
3 **foreach** $v_j \in \hat{\mathcal{M}}\backslash v_i$ **do**
4 $\quad$ **if** v_j receives valid `update` in τ_0+1 **then**
5 $\quad\quad$ v_j sends $(\texttt{acpt}, w+1, \sigma_j)$ to $\hat{\mathcal{M}}\backslash v_j$;
6 $\quad$ **else** v_j sends $(\texttt{rej}, w+2, \sigma_j)$ to $\hat{\mathcal{M}}\backslash v_j$;
7 **foreach** $v_i \in \hat{\mathcal{M}}$ **do**
8 $\quad$ **if** v_i misses `acpt` in $\tau_0 + 2$ **then**
9 $\quad\quad$ v_i registers a dispute;
10 $\quad$ **else if** v_i receives any `rej` in τ_0+2 **then**
11 $\quad\quad$ v_i sends $(\texttt{rej}, w+2, \sigma_i)$ to $\hat{\mathcal{M}}\backslash v_i$;
12 $\quad\quad$ **if** $\forall v_j \in \hat{\mathcal{M}}\backslash v_i$ receives `rej` in τ_0+3 **then**
13 $\quad\quad\quad$ $\hat{\mathcal{M}}$ reaches consensus to reject w+2;
14 $\quad\quad$ **else** v_j registers a dispute;
15 $\quad$ **else** $\hat{\mathcal{M}}$ reaches consensus to update w+1;

E. MPC Operations

Once the MPC validators are selected, they proceed to construct the MPC. Similar to existing virtual payment channel protocols [18, 20, 31], we define the MPC contract as `C`, and the lifetime of an MPC in ShardTree consists of three operations—*construction*, *update*, and *closure*. The ultimate purpose

Algorithm 5: MPC Closure

Input: selected validator set $\hat{\mathcal{M}}$.

1 **foreach** $v_i \in \hat{\mathcal{M}}$ **do**

// unlock MPC balance back to PC

2 Execute close function in C;

of "constructing" an MPC is for every selected validator in $\hat{\mathcal{M}}$ to add an instance of executing C in their subchannels. In the optimistic case, when all MPC validators are honest, they can open/update/close a contract instance within $O(1)$ rounds. Each operation is detailed separately in the following Algorithm 3, Algorithm 4, and Algorithm 5:

MPC Construction: Algorithm 3 describes the procedure for constructing an MPC under the coordination of l_e and all selected validators in $\hat{\mathcal{M}}$, starting from a round τ_0. l_e first broadcasts a construction request message construct containing all related information $(c, \hat{\mathcal{C}}, \hat{\mathcal{M}}, \hat{\mathcal{P}})$ and its signature σ_e (Line 1). Upon receiving the construct message, validator v_i requests an update of C. If $b_i \leqslant b_j$, v_i sends an open message in τ_0 to v_j and executes an instance of C locally by locking corresponding funds according to $\hat{\mathcal{C}}$ (Lines 4 to 5). Otherwise, v_i waits for receiving open message. v_i inspects the received open and construct by checking the respective signatures, correctness of the message, and subchannel identity. If checking passes, v_i executes C locally. v_i sends an ack message back to v_j in τ_0+1 (Line 7). If all parties follow the protocol, in round τ_0+2, all the subchannels of the MPC should have executed C, every v_j sends its opened message to other MPC validators (Lines 8 to 10). In case any validator v_j other than v_i fails to receive an expected opened in τ_0+3, it sends a rej (reject) message to other selected validators and exits (Lines 12 to 14). Likewise, if any v_j receives a rej in τ_0+4, it terminates the process. If v_j receives no rej, the MPC construction is completed, and it sends a created message to l_e (Lines 15 to 17). Algorithm 3 coordinates consistent MPC initialization by handling message delays and faults via timed acknowledgments and rejections.

MPC Update: Algorithm 4 describes the process for updating MPC state under version control. During the consensus phase of ShardTree, an MPC state update can be initiated by any MPC validator of the sender shard. If a validator v_i receives update in τ_0, v_i checks the update by checking the signature of the shard consensus leader, v_i signs and sends back the update message $(\texttt{update}, w, \sigma_j)$ if update is valid (Lines 1 to 2). In the subsequent round, the rest validators $v_j \in \hat{\mathcal{M}} \backslash v_i$ verifies if the received update is valid. If valid, v_j sends an acceptance message $(\texttt{acpt}, w{+}1, \sigma_j)$ on state version w+1 to other validators (Lines 4 to 5). If not valid, a rejection message $(\texttt{rej}, w{+}2, \sigma_j)$ is sent (Line 6). The protocol proceeds with fault detection and consensus. If any validator misses an acpt message in τ_0+2, it registers a dispute (Line 9); otherwise, they reach a consensus on an update (Line 15). We adopt direct on-chain dispute resolution to reduce communication overhead as in [18]. Similarly, if all rejections are received in τ_0+3, a consensus on rejection is reached (Line 13); otherwise, a dispute is registered (Lines 10 to 14). Essentially, these validators have to reach a consensus to either reject or accept an update.

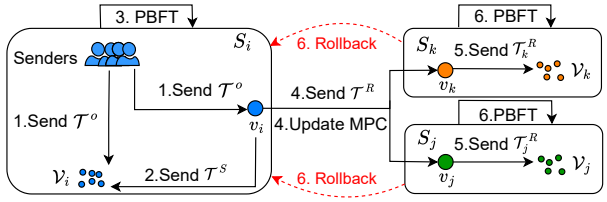


Fig. 5: Cross-shard transaction processing

MPC Closure: Algorithm 5 shows how to close an MPC and release any associated funds back to its subchannels. Any MPC validator can automatically initiate the closure starting at round τ_0, which is reaching the end of an epoch. For simplicity, every MPC validator needs to locally trigger the close function in C that unlocks their corresponding MPC balance to the subchannel according to the last agreed state version w [18].

F. MPC Reconstruction During Epoch

When a malicious MPC validator is detected or all MPC validators in a shard exhaust their capacity share, another MPC can be constructed. The former goes through a dispute on-chain [18], whereas the latter undergoes a peaceful MPC closure and reopening through Algorithm 5 and Algorithm 3.

VI. Cross-shard Transaction Processing

In this section, we introduce how ShardTree handles the cross-shard transactions in the consensus phase.

A. Processing Cross-shard Transactions

ShardTree processes cross-shard transactions by generating three new types of transactions: **S-type**, **MPC**, and **R-type**. An S-type transaction sends funds from a sender to an MPC validator in the sender shard, while an R-type transaction delivers funds from an MPC validator to the respective receiver. A key advantage of the MPC is that only one MPC transaction is needed per batch of cross-shard transactions, regardless of the number of senders or receivers that exist in these cross-shard transactions. Each multi-sender, multi-receiver transaction can be decomposed into a set of single-sender, single-receiver transactions with the same lock duration. For example, if a transaction is split into $(A{\rightarrow}C, 2)$, $(B{\rightarrow}C, 1)$, and $(B{\rightarrow}D, 2)$, with the amounts and recipients predetermined by the users, it represents a two-sender, two-receiver transaction, where senders A and B collectively send a total of 5 tokens to receivers C and D. These one-sender one-receiver decomposed transactions are treated as the *original transactions*. Fig. 5 illustrates the steps for handling cross-shard transactions, with three MPC validators v_i, v_j, and v_k representing shards S_i, S_j, and S_k, respectively. We consider an original transaction set $\mathcal{T}^O$, which is processed from the sender shard S_i (steps 1–3), through the MPC (step 4), and finally to the receiver shards S_j and S_k (steps 5–6).

Step 1: We define each original transaction $T^O \in \mathcal{T}^O$ as a tuple $T^O := (s, t, x, \phi, \sigma_s, H_i, \eta)$, where s and t denote the sender and receiver, respectively, x is the transaction amount, $\phi = \{\phi^s, \phi^t\}$ represents the fees for the two MPC validators in

the sender shard and receiver shard, respectively, computed based on transaction amount of x (see Section III-A). σ_s denotes the ECDSA signature of sender [24], H_i denotes the block height in S_i at the time of the transaction initiation, and η specifies the lock duration, i.e. the number of future blocks starting from H_i for x being locked in S_i. A strictly monotonic nonce is included in T^O to prevent replay attacks [1]. $\mathcal{T}^O$ is routed to all shards (e.g. via Kademlia routing protocol in existing sharding works [9, 12, 38]).

Step 2: The MPC validator v_i generates a corresponding S-type transaction T^S for each T^O, defined as $T^S{:=}(T^O, v_i, \sigma_i)$, where s in the T^O is the sender, v_i is the receiver, and σ_i is the signature of v_i. Next, v_i groups and sends a set of S-type transactions $\mathcal{T}^S$ to rest validators $\mathcal{V}_i$ in shard S_i.

Step 3: Shard S_i proposes $\mathcal{T}^S$ with other intra-shard transactions as a new block, verifies it via PBFT by checking sender balances and signatures, and appends it locally if valid.

Step 4: v_i creates a set of R-type transactions $\mathcal{T}^R$ corresponding to $\mathcal{T}^S$, where each $T^R{\in}\mathcal{T}^R$ is defined as $T^R{:=}(T^S, v_j, H_i, \sigma_i)$ with v_j as the sender, t in T^S as the receiver, H_i as the current block height of S_i, and σ_i as the signature of v_i. v_i sends $\mathcal{T}^R$ to v_j and v_k. v_j extracts $\mathcal{T}_j^R{\subseteq}\mathcal{T}^R$, consisting of transactions whose receivers are accounts in shard S_j. Similarly, v_k extracts $\mathcal{T}_k^R{\subseteq}\mathcal{T}^R$ for shard S_k. Meanwhile, v_i initiates the MPC state update using Algorithm 4, reducing its MPC capacity share based on the net transaction amounts and fees in $\mathcal{T}^R$, while v_j and v_k increase theirs according to $\mathcal{T}_j^R$ and $\mathcal{T}_k^R$, respectively.

Step 5: v_j forwards $\mathcal{T}_j^R$ to the rest validators $\mathcal{V}_j$ in shard S_j. $\mathcal{V}_j$ then verifies each R-type transaction $T^R{\in}\mathcal{T}_j^R$ by checking two conditions: 1) it is received before its lock expires, i.e., $H_j{<}\frac{2}{3}(\eta){+}H_i$, where H_j is the current block height in S_j, η and H_i are the lock duration and block height of S_i in T^O; and 2) the signature σ_i is valid. $\mathcal{V}_k$ does the same for each R-type transaction $T^R{\in}\mathcal{T}_k^R$.

Step 6: If neither condition is met, then S_j or S_k will fail to reach PBFT consensus on either $\mathcal{T}_j^R$ or $\mathcal{T}_k^R$; a rollback scheme is activated to ensure the transaction atomicity (detailed in Section VI-B). Otherwise, T_j^R and T_k^R are committed in S_j and S_k through PBFT consensus, respectively.

B. Atomicity Guarantee

We still use Fig. 5 to explain the rollback scheme. Each validator knows the lock duration for the original transaction from Step 1. Thus, it is easy for the receiver shard to detect if the respective R-type transaction is received on time. Note that MPC updates are significantly faster than PBFT consensus, as they occur off-chain and require agreement only among the MPC validators. To allow sufficient time for the MPC update, we divide the lock duration equally among the two intra-shard consensus and the MPC update. This time allocation is configurable based on system specifications. Without loss of generality, we consider a failed original transaction T^O with its sender in S_i and receiver in S_j. We summarize all three possible failure cases as follows:

Case 1 (Failure of v_i): v_i fails to forward T^R to v_j. Thus, the MPC is not updated, and T^R is not transmitted to S_j. To handle this case, any honest validator in S_j who observes that T^R has not been transmitted within $H_i{+}\frac{2}{3}\eta$ initiates a rollback proof R, where H_i and η are the block height and lock duration in T^O. This honest validator then groups failed cross-shard transactions $\mathcal{R}_i$ whose sender shards are all S_i and uses them to generate R, where $T^O{\in}\mathcal{R}_i$. Let $\mathsf{R}{:=}(\mathcal{R}_i, v_i, H_j, \sigma)$ denote the rollback proof for shard S_i, where $\mathcal{R}_i$ is a set of rollback transactions, H_j is the current block height of S_j when R is initiated, v_i is the sender responsible for paying all transactions in $\mathcal{R}_i$, and σ denotes the signature of the rollback proof initiator. Next, the honest validator broadcasts R to $\mathcal{V}_j$. By checking if $H_j \geqslant H_i{+}\frac{2}{3}\eta$ and if σ is valid, S_j reaches a PBFT consensus and any honest validator in S_j broadcasts R to S_i; otherwise, R is aborted. After S_i receives R, S_i also runs a PBFT consensus on R. To exclude v_i from the MPC, the remaining MPC validators can initiate an MPC closure via Algorithm 5 and reconstruct an MPC via Algorithm 2 and Algorithm 3 (See Section V-F). Note that the reconstruction of MPC requires no on-chain operation through peaceful closure.

Case 2 (Failure of v_j): v_j fails to forward T^R to $\mathcal{V}_j$. The MPC has been updated, and $\mathcal{V}_j$ does not receive T^R. The rollback proof in this case is the same R as above. Any honest validator in S_j follows the same process to validate R as in Case 1. In Case 2, the MPC state must also be rolled back. v_i forces a new MPC update after receiving R as in Algorithm 4.

Case 3 (Collusion between MPC validators): v_i and v_j collude, where the MPC remains unchanged, and T^R is not transmitted to S_j. MPC validators cannot all benefit from collusion, as at least one validator lies on the opposite side of the other validators. The MPC capacity remains the same as when the channel was initially opened if no new deposits are made [19]. Thus, Case 3 is the least likely to happen. If Case 3 still occurs, the rollback scheme and an MPC can be reconstructed similar to that in Case 1. MPC also ensures unilateral closure even if there is only one honest MPC validator (see Theorem 3).

VII. Security Analysis

In this section, we analyze the security of ShardTree in terms of shard-size, epoch, and MPC properties. Our analysis builds on Byzantine fault tolerance and probabilistic guarantees to demonstrate security against adversarial behavior.

A. Shard-Size Security

Similar to prior sharding protocols [3–5, 9, 10], we prove the safety and liveness of ShardTree in Theorem 2. Safety ensures honest validators agree on the same valid block in each consensus, while liveness guarantees that every proposed block is eventually committed or aborted.

Theorem 2. ShardTree *ensures transaction processing safety and liveness if each shard contains $f{<}\frac{n}{3}$ malicious validators.*

Proof. Following [2, 3, 5, 9, 11, 12], PBFT consensus requires a shard size of $n{=}3f{+}1$ to provide a valid consensus result, and the security of shard assignment is modeled as a random sampling problem using the cumulative hypergeometric distribution: $\Pr[X{\geqslant}\lfloor\frac{n}{3}\rfloor]{=}\sum_{i=\lfloor\frac{n}{3}\rfloor}^{n}\frac{\binom{\mathcal{F}}{i}\binom{N-\mathcal{F}}{n-i}}{\binom{N}{n}}$, where X represents

a random variable denoting the number of corrupt validators in the shard, N is the number of validators in the network, and $\mathcal{F}$ is the fraction of corrupted validators in the whole network. For example, when N=600, n=100 and $\mathcal{F}$=100, $\Pr[X \geqslant \lfloor \frac{n}{3} \rfloor] \approx 6.82e^{-6}$, meaning that a shard might fail once in 402 years with repeating one-day epochs. All system messages are secured against forgery through digital signatures as shown in Section IV. Thus, honest validators in ShardTree agree on the same newly proposed block, which ensures safety.

In addition, validators are connected by a partially synchronous network in ShardTree. If an honest consensus leader proposes a valid block, it will eventually be broadcast and received by a majority of validators through PBFT consensus. If the receiver shard does not receive the cross-shard transactions within the lock duration, at least one validator in the receiver shard will initiate the rollback scheme to abort the false transactions as in Section VI-B. Thus, no malicious validators can indefinitely halt the consensus phase, and every block will eventually be committed or aborted. □

B. Epoch Security

The failure probability of a single shard does not fully reflect the failure probability of the entire sharding system. A system failure occurs if at least one shard fails [3, 5, 9, 39]. Following the cumulative hypergeometric distribution function [5, 9], we formulate the upper bound of the system failure probability in an epoch as: $Pr[Fail_s] = 1 - \left[1 - \sum_{i=\lfloor \frac{n}{3} \rfloor}^{n} \frac{\binom{\mathcal{F}}{i}\binom{N-\mathcal{F}}{n-i}}{\binom{N}{n}}\right]^{M} < 2^{-\lambda}$, where λ is the preset security parameter for ShardTree. System failure is negligible by adjusting the size of the network N, and the number of shards M [2, 3, 5, 9, 39].

C. MPC Security

First, MPC validators are incentivized to act honestly due to the fee mechanism in ShardTree, which ensures them rewarded for honest transaction delivery. Second, Theorem 3 proves the correctness and liveness of MPC in ShardTree. Correctness ensures that no validator can claim more balance than the last agreed MPC state, and liveness ensures that if all but one validator disappears, the remaining validator can unilaterally close the channel using the latest state.

Theorem 3. ShardTree *guarantees MPC correctness and liveness.*

Proof. In ShardTree, since every valid state is jointly signed by all MPC validators, a single validator cannot fabricate a state that records the funds of every validator without the others' signatures. Therefore, unilateral theft of funds is impossible. This guarantees the MPC correctness in ShardTree.

To prove the MPC liveness, we consider two scenarios: 1) When an epoch ends, MPC smart contract allows any honest validator to enforce automatic termination and claim their balance without other MPC validators; 2) If an older state is submitted, any honest MPC validator can initiate a dispute on-chain [18, 21, 22], which holds even if all other validators in the MPC collude. This proves the liveness of MPC. □

VIII. Performance Evaluation

A. Implementation and Evaluation Setup

To evaluate the performance of ShardTree, we implemented it on an open-sourced blockchain testbed, BlockEmulator [40]. Each simulation hardware is equipped with 9-core processors, 32GB of RAM, and a 10 Gbps network connection. Validators communicate through a peer-to-peer (P2P) network. Each validator independently runs the same decentralized protocol to process incoming transactions. We sampled 1,000,000 Ethereum transactions in chronological order from block height 1,000,000 to 1,999,999 [41]. We also sampled 10,000 account balances from Ethereum accounts [42]. We repeated 30 times for each setting to average out the network noise.

B. MPC Construction Evaluation

	Lat. (s)	Lat. percentage (%)	Capacity (10^8 wei)
ShardTree	0.026	0.030	1.47

Table II: MPC construction evaluation results. Latency is abbreviated as Lat.

We evaluate the MPC construction in ShardTree using metrics: latency, latency percentage, and capacity. Latency measures the average time taken for MPC construction, and the latency percentage represents its proportion relative to an epoch. Table II shows that ShardTree achieves a low latency of 0.026 seconds, a low latency percentage of 0.030 %, and a high capacity, indicating very low construction delay. This demonstrates the efficiency of the MPC construction.

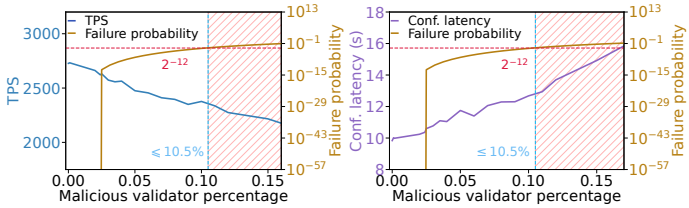
C. Transaction Processing Evaluation

We used the following metrics to assess the transaction processing of ShardTree: transactions per second (TPS), confirmation latency (s) (CL), queue size, and injection size. TPS is the number of transactions processed divided by the total processing time. CL is the average time taken for a transaction to be confirmed. TPS is about how many transactions the system handles in a given time frame, whereas CL is about how fast each transaction is processed. Thus, high TPS does not inherently lead to low CL, or vice versa. The queue size represents the number of transactions in the transaction pool, reflecting how quickly transactions can be accumulated. The injection size represents the number of transactions injected into the network, indicating how quickly the transaction pool can be saturated. We examine under default values M=16 and N=160, where M is the number of shards, and N is the total number of validators. We compare ShardTree with two state-of-the-art sharding protocols:

- Monoxide [43]: The sender shard generates and forwards relay transactions for cross-shard transactions using a dual-stage handling mechanism.
- BrokerChain [1]: A fixed number of brokers, each maintaining accounts across multiple shards, are responsible for handling cross-shard transactions on behalf of all shards. Each cross-shard transaction is decomposed into two sub-transactions, referred to as Type1 and Type2.

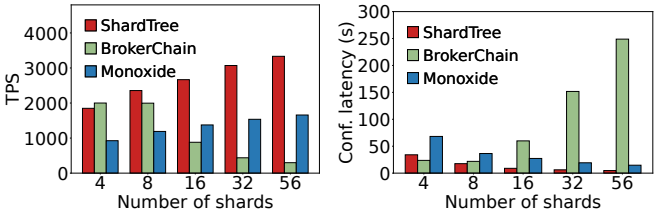
D. Evaluation Results

Impact of Malicious Validator Percentage: Fig. 6(a) and Fig. 6(b) illustrate the performance of ShardTree in terms of



(a) TPS (b) Confirmation latency

Fig. 6: Impact of malicious validator percentage



(a) TPS (b) Confirmation latency

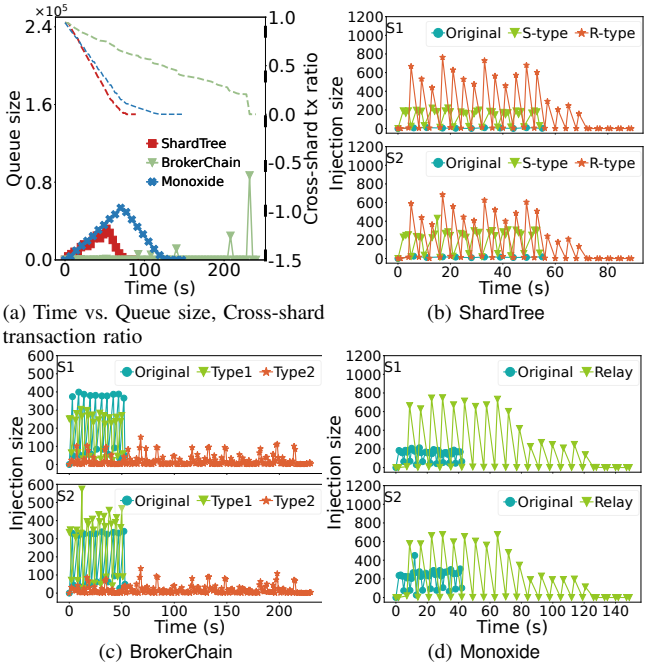
Fig. 7: Impact of M

TPS and CL with varying malicious validator percentages. To simulate malicious validators, a PBFT block proposer actively participates in generating and broadcasting false blocks to disrupt the consensus, which also results in false MPC updating. As the malicious validator percentage increases, the TPS decreases, CL increases, and failure probability increases. These trends occur because the false blocks proposed by malicious validators are aborted through PBFT. To satisfy $Pr[Fail_s]<2^{-12}$, which indicates one failure in ∼157 years for two-week epochs (computed through Section VII-B), the malicious validator percentage needs to remain below 10.5%.

Impact of M: Fig. 7 illustrates the performance of the three protocols in terms of TPS and CL with respect to M. ShardTree outperforms the other two protocols as M grows for both metrics. In Fig. 7(a), the TPS increases with M for both ShardTree and Monoxide. In Fig. 7(b), the CL decreases with M for both ShardTree and Monoxide. As M increases, more shards can process transactions in parallel, resulting in a positive effect on TPS and CL. We observe a reverse trend of TPS and CL for BrokerChain. The brokers deployed in BrokerChain for processing cross-shard transactions quickly become the bottleneck of this process when the number of cross-shard transactions exceeds their processing ability, which occurs as M increases.

Impact of Queue Size: Fig. 8(a) shows the queue size and cross-shard transaction ratio of the three protocols over time, respectively. When transactions are generated, the queue size initially increases and then decreases to zero over time for all three protocols. The queue size in ShardTree grows slowly and drops more rapidly than in the other two protocols, indicating lower congestion in the transaction pool, which is further supported by the faster decline in the cross-shard transaction ratio. Although BrokerChain does not show early transaction accumulation, its longer processing time results in lowest TPS.

Impact of Injection Size: Fig. 8(b), Fig. 8(c), and Fig. 8(d) illustrate how injection size changes of different transaction types over time for the three protocols within S_1 and S_2. It is observed that all original transactions are injected within ∼50



(a) Time vs. Queue size, Cross-shard transaction ratio (b) ShardTree (c) BrokerChain (d) Monoxide

Fig. 8: Comparison results for ShardTree, BrokerChain and Monoxide

seconds across three protocols, whereas S-type, Type2, and Relay transactions are distributed over the entire processing period. This is because the latter transactions are created after the original ones and are processed in the order they arrive. BrokerChain takes longer to inject and process its types of transactions, resulting in a lower TPS. In addition, ShardTree shows a similar pattern for R-type transactions as Relay transactions in Monoxide. This is because both protocols use the sender shards to process the converted transactions. However, ShardTree utilizes MPC to reduce overhead for broadcasting R-type transactions, and thus uses less time to finish transaction injection compared to Monoxide.

IX. Conclusion

We proposed a cross-shard protocol, ShardTree, that uses an MPC to connect all shards for handling cross-shard transactions. We first studied the path-based MPC validator selection with the maximum capacity. We also formally designed the MPC operations across shards. We further designed a rollback scheme and provided formal security proofs to ensure the atomicity and security of the ShardTree, respectively. Evaluations demonstrated that ShardTree outperforms other sharding protocols in transaction throughput and confirmation latency with minimized MPC construction overhead.

Future Work: ShardTree was designed for payments, but can be extended to support smart contracts by deploying multi-party virtual state channels. **State channels** extend PCs to support arbitrary off-chain state transitions, enabling secure off-chain execution of smart contracts. Users can interact with on-chain smart contracts and execute state transitions off-chain through state channels. MPCs are useful for scenarios that are not only for payments, such as online gaming and digital asset exchanges through multi-party smart contracts. We leave this more involved topic for future work.

REFERENCES

[1] 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 *IEEE INFOCOM*, 2022, pp. 1968–1977.

[2] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *SIGSAC*. ACM, 2016, pp. 17–30.

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

[4] W. Chen, D. Xia, Z. Cai, H.-N. Dai, J. Zhang, Z. Hong, J. Liang, and Z. Zheng, "Porygon: Scaling blockchain via 3d parallelism," in *IEEE ICDE)*, 2024, pp. 1944–1957.

[5] Z. Hong, S. Guo, P. Li, and W. Chen, "Pyramid: A layered sharding blockchain system," in *IEEE INFOCOM*, 2021, pp. 1–10.

[6] F. Cheng, J. Xiao, C. Liu, S. Zhang, Y. Zhou, B. Li, B. Li, and H. Jin, "Shardag: Scaling dag-based blockchains via adaptive sharding," in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2024, pp. 2068–2081.

[7] S. Jiang, J. Cao, C. L. Tung, Y. Wang, and S. Wang, "Sharon: Secure and efficient cross-shard transaction processing via shard rotation," in *IEEE INFOCOM*, 2024, pp. 2418–2427.

[8] J. XU, "xshard," 2024. [Online]. Available: https://github.com/myl7/xshard/

[9] M. Li, Y. Lin, J. Zhang, and W. Wang, "Cochain: High concurrency blockchain sharding via consensus on consensus," in *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*, 2023, pp. 1–10.

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

[11] X. Wang, B. Li, L. Jia, and Y. Sun, "Orbit: A dynamic account allocation mechanism in sharding blockchain system," in *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2024, pp. 333–344.

[12] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," in *2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 931–948.

[13] Hyperledger Fabric Team, "Hyperledger Fabric: What is Hyperledger Fabric?" 2022. [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.2

[14] Z. Team, "ZILLIQA 2.0 Whitepaper," 2025. [Online]. Available: https://www.zilliqa.com/

[15] Tendermint Team, "Tendermint: BFT Consensus for Blockchains," 2024. [Online]. Available: https://tendermint.com/

[16] M. Du, Q. Chen, and X. Ma, "Mbft: A new consensus algorithm for consortium blockchain," *IEEE Access*, vol. 8, pp. 87 665–87 675, 2020.

[17] G. G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu, "Sbft: A scalable and decentralized trust infrastructure," in *IEEE/IFIP DSN*, 2019, pp. 568–580.

[18] S. Dziembowski, L. Eckey, S. Faust, J. Hesse, and K. Hostáková, "Multi-party virtual state channels," in *Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part I 38*. Springer, 2019, pp. 625–656.

[19] E. Foundation, "What is the raiden network," 2019. [Online]. Available: https://raiden.network

[20] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments," 2016.

[21] S. Dziembowski, S. Faust, and K. Hostáková, "General state channel networks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 949–966.

[22] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski, "Perun: Virtual payment hubs over cryptocurrencies," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 106–123.

[23] Q. Wei, D. Yang, R. Yu, and G. Xue, "Thor: A virtual payment channel network construction protocol over cryptocurrencies," in *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*. IEEE, 2024, pp. 771–780.

[24] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)," *International journal of information security*, vol. 1, pp. 36–63, 2001.

[25] W. Li, C. Feng, L. Zhang, H. Xu, B. Cao, and M. A. Imran, "A scalable multi-layer pbft consensus for blockchain," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1146–1160, 2020.

[26] S.-E. Huang, S. Pettie, and L. Zhu, "Byzantine agreement in polynomial time with near-optimal resilience," in *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, 2022, pp. 502–514.

[27] K. Ren, N.-M. Ho, D. Loghin, T.-T. Nguyen, B. C. Ooi, Q.-T. Ta, and F. Zhu, "Interoperability in blockchain: A survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 12, pp. 12 750–12 769, 2023.

[28] C. Boyd, K. Gjøsteen, and S. Wu, "A blockchain model in tamarin and formal analysis of hash time lock contract," in *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 5–1.

[29] Y. Zhang and D. Yang, "Robustpay+: Robust payment routing with approximation guarantee in blockchain-based payment channel networks," *IEEE/ACM Transactions on Networking*, vol. 29, no. 4, pp. 1676–1686, 2021.

[30] X. Wang, R. Yu, D. Yang, G. Xue, H. Gu, Z. Li, and F. Zhou, "Fence: Fee-based online balance-aware routing in payment channel networks," *IEEE/ACM Transactions on Networking*, vol. 32, no. 2, pp. 1661–1676, 2023.

[31] "Raiden network," 2025. [Online]. Available: https://raiden.network/

[32] G. Di Stasi, S. Avallone, R. Canonico, and G. Ventre, "Routing payments on the lightning network," in *2018 IEEE international conference on internet of things (IThings) and IEEE green computing and communications (GreenCom) and IEEE cyber, physical and social computing (CPSCom) and IEEE smart data (SmartData)*. IEEE, 2018, pp. 1161–1170.

[33] G. Karypis and V. Kumar, "Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," 1997.

[34] E. Fynn and F. Pedone, "Challenges and pitfalls of partitioning blockchains," in *2018 48th annual IEEE/IFIP international conference on dependable systems and networks workshops (DSN-w)*. IEEE, 2018, pp. 128–133.

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

[36] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th symposium on operating systems principles*, 2017, pp. 51–68.

[37] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," in *International workshop on peer-to-peer systems*. Springer, 2002, pp. 53–65.

[38] J. Xu, Y. Ming, Z. Wu, C. Wang, and X. Jia, "X-shard: Optimistic cross-shard transaction processing for sharding-based blockchains," *IEEE Transactions on Parallel and Distributed Systems*, 2024.

[39] L. N. Nguyen, T. D. Nguyen, T. N. Dinh, and M. T. Thai, "Optchain: optimal transactions placement for scalable blockchain sharding," in *IEEE ICDCS*, 2019, pp. 525–535.

[40] HuangLab-SYSU, "Block emulator," 2024. [Online]. Available: https://github.com/HuangLab-SYSU/block-emulator

[41] X. Team, "Xblock," 2024. [Online]. Available: https://xblock.pro/xblock-eth.html

[42] Etherscan, "Ethereum accounts by balance," https://etherscan.io/accounts, 2025, accessed: 2025-07-22.

[43] J. Wang and H. Wang, "Monoxide: Scale out blockchains with asynchronous consensus zones," in *USENIX NSDI*, 2019, pp. 95–112.