Kronos: A Robust Sharding Blockchain Consensus with Optimal Communication Overhead

Andi Liu, Yizhong Liu, Zhuocheng Pan, Yinuo Li, Jianwei Liu, Yuan Lu

ABSTRACT

Sharding enhances blockchain scalability by dividing the network into shards, each managing specific unspent transaction outputs or accounts. As an introduced new transaction type, cross-shard transactions pose a critical challenge to the security and efficiency of sharding blockchains. Current solutions, however, either prioritize security with assumptions and substantial investments, or focus on reducing overhead and overlooking security considerations.

In this paper, we present Kronos, a *generic* and efficient sharding blockchain consensus ensuring *robust* security. At the core of Kronos, we introduce a "buffer" mechanism for *atomic* cross-shard transaction processing. Shard members collectively maintain a buffer to manage cross-shard inputs, ensuring that a transaction is committed only if all inputs are available, and no fund is transferred for invalid requests. While ensuring security including atomicity, Kronos processes transactions with *optimal* intra-shard communication overhead. A valid cross-shard transaction, involving x input shards and y output shards, is processed with a minimal intra-shard communication overhead factor of $x + y$. Additionally, we propose a reduction for transaction invalidity proof generation to simple and fast multicasting, leading to atomic rejection without executing full-fledged Byzantine fault tolerance (BFT) protocol in optimistic scenarios. Moreover, Kronos adopts a newly designed "batch" mechanism, reducing inter-shard message complexity for cross-shard transactions from $O(\lambda)$ to $O(\frac{m\log m}{b}\lambda)$ without sacrificing responsiveness (where m denotes number of shards, b denotes the batch size of intra-shard consensus, and λ is security parameter).

Kronos operates without dependence on any time or client honesty assumption, serving as a plug-in sharding blockchain consensus supporting applications in diverse network environments including *asynchronous* ones. We implement Kronos using two prominent BFT protocols: asynchronous Speeding Dumbo (NDSS'22) and partial synchronous HotStuff (PODC'19). Extensive experiments (over up to 1000 AWS EC2 nodes across 4 AWS regions) demonstrate Kronos achieving a substantial throughput of 68.6ktx/sec with 1.7sec latency. Compared with state-of-the-art solutions, Kronos outperforms in all cases, achieving up to a 42× improvement in throughput and a 50% reduction in latency when cross-shard transactions dominate the workload.

CCS CONCEPTS

• **Security and privacy → Distributed systems security**.

KEYWORDS

Blockchain, Sharding Consensus, Asynchronous Atomicity, Byzantine Fault Tolerance

1 INTRODUCTION

Blockchain technology has attracted widespread attention since its inception alongside Bitcoin [44]. It leverages fault-tolerant consensus and cryptographic technologies to facilitate a distributed ledger. Owing to its exceptional properties of security, decentralization, and transparent storage, blockchain has become a valuable choice not only for cryptocurrencies, including well-known examples like Bitcoin and Ethereum, but also for applications across diverse industries such as financial services, the Internet of Things (IoT), supply chain management. Various state-of-the-art researches, such as federal learning, privacy-preserving computation, and identity authentication [13, 20, 21, 34, 46], incorporate blockchain technology to improve overall performance. The properties of blockchain also drive the development of emerging domains such as Web3.0 [4, 35] and Metaverse [30, 43].

Scalability bottleneck and the potential of sharding. Practical applications reveal a significant challenge in traditional blockchains—*poor scalability* [41]. Every transaction requires submission to the entire network for consensus, and participants verify each transaction through a *Byzantine fault tolerance* (BFT) protocol. As the number of participants grows, the heightened verification overhead results in a reduction in throughput.

Elastico [41] first proposes *sharding blockchain* to alleviate the problem through the lens of sharding technology (which originates from the database field). The sharding paradigm partitions all parties into a few smaller groups, referred to as *shards*, and each shard's majority of the workload is to process a subset of transactions. Hence, each party only has to participate in some small shards instead of interacting with the entire network, preserving low overhead despite the total number of nodes in the system. Moreover, when a sharding blockchain is scaled to contain more participants and shards, it is even promising to achieve higher transaction throughput, as multiple shards are expected to process different transactions in parallel [38]. Currently, significant attention has been directed towards blockchain sharding, with notable examples including [3, 10, 15, 23, 28, 29, 33, 37, 53, 55].

Introduced a new scenario: cross-shard transactions. Sharding technology presents a promising solution for enhancing blockchain scalability, yet it introduces specific challenges. In sharding blockchains, each transaction undergoes processing solely by one (or more) of all shards, enabling multiple transactions to be processed in parallel. It is crucial to ensure that a majority of participants in each shard are honest, which means the proportion of Byzantine nodes falls within the fault tolerance of the adopted BFT protocol. Therefore, the methods for secure shard configuration become a pivotal area of research, and many prominent works, such as Gearbox [17] (CCS'22), provide remarkable solutions.

Another critical challenge is *cross-shard transaction* processing. Each shard separately manages a part of addresses according to specific assignment rules along with the *unspent transaction outputs* (UTXOs)/accounts associated with them, bringing in a transaction type where input and output addresses belong to different shards. These transactions are called *cross-shard transactions* [33]. Shards responsible for managing certain input UTXOs (or accounts) are called *input shards* of the transaction, and shards receiving transaction outputs are called *output shards* (respectively denoted as S_{in} and S_{out} in this paper). Due to the state isolation across shards, cross-shard transactions cannot be processed by a single shard solely but require multiple shard cooperation.

Cross-shard transaction processing demands critical attention. An inappropriate mechanism not only hampers efficiency but also ruins overall system security if a cross-shard transaction is processed inconsistently across involved shards [36].

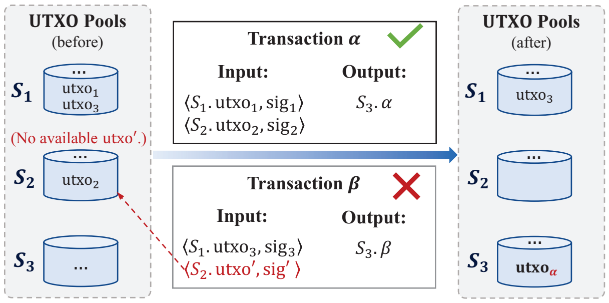


Figure 1: Cross-shard transaction atomic execution.

Necessity of secure processing with atomicity. In addition to the fundamental requirements of *persistence* and *consistency* for secure blockchains, ensuring *atomicity* in transaction processing emerges as a crucial property. Atomicity guarantees that a transaction execution is "all-or-nothing", where all operations across all involved shards commit, or every operation is aborted. As exemplified in Figure 1, the inputs of transaction α are both available (utxo$_1$ and utxo$_2$ with verified signatures sig$_1$ and sig$_2$), requiring "all" execution, where utxo$_1$ and utxo$_2$ are both spent for tx_α. The failure to spend any inputs for a committed transaction compromises blockchain security, leading to an imbalance where the output value exceeds the actual inputs and a risk of double-spending.

In case a transaction requests to spend an *unavailable* input (with non-existent utxo, missing or invalid client signature), such as transaction β in Figure 1, it is deemed invalid and shards operate "nothing". No utxo will be transferred to the output, even though it might be available (e.g., utxo$_3$ remains in shard S_1). Any execution of an invalid transaction, whether it results in committing the total output value or transferring available inputs partially, is detrimental. The former compromises ledger security, while the latter leads to losses for clients, especially in cases where the invalid transaction is intentionally crowdfunded by a malicious client paying nothing.

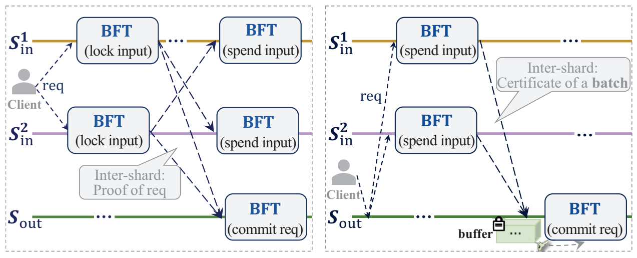
Necessity of efficient processing. Cross-shard transaction processing efficiency significantly impacts the overall system performance. Research [53] indicates that cross-shard transactions constitute a substantial proportion in sharding blockchain systems, and their occurrence tends to rise with an increased shard number. Improper handling might undermine the anticipated performance gains associated with a larger shard number.

The cost of cross-shard transaction processing primarily includes two aspects: *intra-shard overhead* and *inter-shard overhead* (also referred to as *cross-shard overhead* in this paper). Intra-shard overhead is the cost incurred by each shard to process a transaction, and the final consistent decision across shards is achieved through inter-shard cooperation. A critical research objective is to minimize both the intra- and inter-shard communication overheads for cross-shard transaction processing without compromising security.

1.1 Remaining issues of prior solutions

While numerous advanced sharding blockchain systems have put forth some methods to address the cross-shard processing challenges, each falls critically short in some aspect.

Two-phase commit: cumbersome overhead and potential atomicity loss. The pioneering works of Omniledger [33] and Chainspace [3] propose committing cross-shard transactions through two phases: *prepare* and *commit*, known as *two-phase commit* (2PC). In the prepare phase, each input shard executes a BFT consensus to either lock available inputs or prove an unavailable input, both with a certificate sent to other involved shards. In the commit phase, each input shard executes BFT again, spending or unlocking the locked inputs based on transaction validity.



(a) 2PC (b) Kronos

Figure 2: Brief Procedures of 2PC and Kronos**.**

Figure 2a illustrates the brief process of 2PC. The locking mechanism ensures that each shard atomically processes received cross-shard transactions but requires executing BFT protocol twice, ensuring consistent state update of inputs (i.e., locked and unlocked/spent). This twofold increase in overhead (compared to normal transaction processing with one BFT protocol execution) leads to efficiency reduction. Moreover, the locked inputs undergo further processing only after receiving certificates from all other input shards. This unlocking approach fails to guarantee liveness in scenarios involving *malicious clients*, who might deliberately submit no request to some input shard. In this case, input shards never receive enough certificates, so honest clients' funds will be "frozen" indefinitely, compromising system liveness.

Follow-up works: over-tilted trade-off between efficiency and security. Subsequent studies [49, 53] recognized the high overhead in 2PC. They eliminate the second BFT protocol executed by input shards and directly spend available inputs to the payee through an intra-shard consensus, regardless of other input states. While this removal reduces input shard overhead, it compromises security by lacking careful operations on the spent inputs, potentially leading to non-atomic execution of invalid transations.

Some works [27, 28] address the security limitations of 2PC relying on honest clients, by introducing another shard serving as a "coordinator" maintaining multiple shard ledgers and proposing cross-shard transactions. While this approach enhances security against malicious clients, the additional communication, computation, and storage in bridge shards hinder full sharding.

A long-neglected but significant problem on inter-shard communication. Inter-shard communication is an indispensable part of cross-shard transaction processing. A final commitment decision requires the availability certificate of every input. In existing sharding blockchains, each shard transmits one certificate to each related shard for one transaction processing. Consider the simplest cross-shard transaction involving 1 input shard and 1 output shard, the communication is seemingly tolerable. If there are b transactions related to the two shards, the inter-shard communication overhead increases to b between the two shards. Expanding to universal scenarios that each transaction involves x input shards and y output shards (a.k.a. *x-in-y-out* in this paper), the overall communication overhead increases to xyb! Such a cost severely impacts cross-shard transaction processing efficiency, and might even influence the intra-shard consensus speed. Some studies [33, 53] suggest packaging multiple input certificates in a single message. While this approach mitigates inter-shard communication complexity to some extent, the overall message complexity remains unchanged.

The above issues expose an open question lying in the design space of sharding blockchains:

Can we present a sharding blockchain consensus achieving practical efficiency while ensuring robust security with atomicity?

1.2 Our contributions

We affirmatively address the aforementioned question by introducing Kronos, a sharding blockchain consensus that ensures security, including atomicity, and optimal overhead.

Robust asynchronous security tolerating malicious client. Kronos achieves secure processing for both valid and invalid transaction requests with atomicity. For a valid transaction request, each available input is spent by the input shard and transferred to the final payee eventually. Any invalid transaction gets comprehensively rejected with an invalidity proof. Spent inputs, rather than being directly transferred to the payee, are temporarily stored in the output shard buffer which is maintained collaboratively by all shard members. Only when every input is received, output shard commits the cross-shard transaction with a quorum proof. Kronos processing methodology accommodates client dishonesty and is not dependent on any time assumptions, guaranteeing security in any network, including poor asynchronous networks, with potential malicious client intervention. As far as we know, Kronos is the first sharding protocol to achieve all of the above properties.

Optimal intra-shard communication overhead. We introduce a metric, *intra-shard communication overhead factor* (IS-COF), to quantify the overall communication cost within each shard for processing a cross-shard transaction, and conduct a thorough analysis of its lower bound. We demonstrate that Kronos achieves the minimum IS-COF for both valid and invalid request processing. For a valid *x-in-y-out* request, the execution is secure with a single BFT protocol execution in each involved shard (while Figure 2b briefly shows), resulting in IS-COF= $x+y$. In the case of an invalid request, proof of invalidity is created through a simple and fast intra-shard multicasting process, significantly reducing the overhead compared with executing a full-fledged BFT protocol in prior solutions. The reduced cost of handling exceptions allows more resources to be allocated to valid transactions.

Efficient inter-shard cooperation. Kronos addresses the challenge of high cross-shard communication overhead (CS-COF) and message overhead (CS-MO) by introducing a batch certification mechanism to prove multiple input availability within a single certificate. Following an intra-shard BFT consensus committing multiple input spending, the shard furnishes each relevant shard with a certificate constructed using Merkle tree technology. Remarkably, only one certificate for each shard is required, regardless of the actual number of inputs to be proven. The batch certificate mechanism leverages the advantages of transaction batch commitment intra-shard, resulting in a decrease in CS-MO from $O(\lambda)$ in previous solutions to $O(\frac{m\log m}{b}\lambda)$ for a cross-shard transaction processing (λ denotes the security parameter), where $m\log m \ll b$ is commonly satisfied in sharding blockchain settings. This release in inter-shard cooperation achieves more efficient cross-shard transaction processing without compromising security.

An implementation of the generic solution. Kronos imposes no restrictions on the BFT protocols employed in each shard and does not rely on any time assumption. As a result, it serves as a generic and plug-in consensus protocol suitable for most sharding blockchains, compatible with various BFT protocols. To demonstrate the practical performance of Kronos, we implement it using an asynchronous BFT, Speeding Dumbo [25], as the exemplary intra-shard BFT consensus, and conduct extensive experiments on Amazon EC2 c5.4xlarge instances distributed from 4 different regions across the globe. The experimental results reveal that Kronos achieves a throughput of 68.6ktx/sec with a network size of 1000 nodes, and the latency is 1.7 seconds. We compare Kronos with 2PC similarly adopting the same BFT protocol. Kronos achieves up to 42× throughput improvement, with a halved latency. To demonstrate the generality, we also deploy Kronos with a partial synchronous BFT HotStuff and evaluate the performance.

2 CHALLENGES AND OUR SOLUTION

Cross-shard transaction processing faces primary challenges in atomicity regarding security, and practical efficiency related to performance. Table 1 provides an overview of the performance of state-of-the-art sharding blockchains. The quantitative metrics in "Cross-Shard Transaction Processing" pertain to *x-in-y-out* cross-shard transactions. k represents the number of input shards managing unavailable inputs. Each protocol in this table (except Elastico, which has no cross-shard transactions) is associated with two sets of indicators for cross-shard transactions, reflecting the processing metrics for valid and invalid requests, respectively. The metrics for *valid* transaction processing are presented without highlighting, and those for *invalid* transaction processing are marked in gray, with k ranging from 1 to x. "MC-Atomicity" signifies atomicity in situations involving malicious clients.

Challenge 1: Robust security with atomicity in asynchronous networks under malicious clients. As aforementioned, secure cross-shard transaction processing is a necessity for sharding blockchain systems. 2PC adopts a locking mechanism, where input shards first lock available inputs and spend them only if all inputs are proven available. This mechanism, however, only considers the circumstance where requests are completely submitted to

Table 1: Comparison of Kronos with state-of-the-art sharding blockchain protocols

Protocol	Network[†]	Throughput	Latency	Cross-Shard Transaction Processing[‡]				
				Solution	IS-COF	CS-COF[§]	CS-MO	MC-Atomicity
Elastico [41]	p. sync.	40 tx/sec	800 sec	—[♭]	—	—	—	—
Omniledger [33]	p. sync.	3500 tx/sec	63 sec	2PC	$2x+y$ $2x-k$	$x(x+y-1)$ $x(x-1)$	$O(\lambda \log b)$	✗
Chainspace [3]	p. sync.	75 tx/sec	15 sec	2PC	$2x+y$ $2x$	$x(x+y-1)$ $x(x-1)$	$O(\lambda)$ (no batch)	✗
RapidChain [53]	sync.	4220 tx/sec	8.5 sec	Splitting	$x+y$ $x-k$	$xy+x+y-1$ $(x-k)y+x+y-1$	$O(\lambda)$ (insecure)	✗
Monoxide [49]	async.	500 tx/sec	90 sec	Relay	$x+y$ $x-k+y$	$x(x+y-1)$ $(x-k)(x+y-1)$	$O(\lambda+\log b)$	✗
Kronos	**sync./ p. sync./async.**	**68600 tx/sec**	**1.7 sec**	**Buffer & Batch**	$\mathbf{x+y}$ $\mathbf{0}$[◇]	$\mathbf{xy+x+y-1}$ $2(x+y-1)$	$O(\frac{m\log m}{b}\lambda)$ [♮]	✓

† "sync.", "p.sync.", and "async." are abbreviations of synchronous, partial synchronous, and asynchronous networks, respectively.
‡ λ, b, and m respectively denote the security parameter, batch size of intra-shard consensus, and the number of shards within the system.
♭ Elastico realizes sharded computation only. Transactions are still recorded in the same ledger, so there are no cross-shard transactions in Elastico.
§ The metrics of CS-COF are computed in *non-client-driven* mode to exclude the influence of client withholding message maliciously.
♮ The relationship $O(\frac{m\log m}{b}\lambda) \ll O(\lambda)$ is commonly satisfied in sharding blockchain settings (e.g., $m = 50$ and $b = 10$k).
◇ The shown IS-COF and CS-COF of Kronos processing invalid transactions are results in common cases. The worst-case values are IS-COF= $2(x-k)$ and CS-COF= $(k+1)(x+y-1)$. See detailed analysis in Section 5 and Appendix C.

every involved shard, i.e., clients are honest. If the client intentionally does not send the request to some input shards, the blinded shards never generate any certificate for the transaction. While other shards are unknowing, they lock their available inputs and wait for responses from other input shards. Obviously, they will never receive enough certificates for a decision, and the available inputs are locked, leading to liveness and atomicity being destroyed.

A trivial solution is to add a timer for each locked input. If certificates of all inputs have not been received when there is a timeout, the transaction is considered to be invalid and shards unlock the inputs automatically. However, this treatment also crashes when the network is asynchronous. In an asynchronous network, a long-time-non-received message cannot correspond to an invalid transaction certainly. Shards cannot distinguish whether the losing certificate is delayed due to a terrible network (i.e. will be received eventually), or the input shard suffers a non-submission by a malicious client, resulting in never creating the required certificate. If an availability certificate arrives at other input shards after a timeout, the transaction cannot be further committed even if it is valid, because other inputs have already been unlocked. The delayed available input remains locked indefinitely, posing a threat to system security. Therefore, the problem of compromising security in asynchronous environments within malicious clients still remains.

Challenge 2: Alleviating heavy consensus overhead without compromising security. Balancing security and efficiency is a crucial consideration. In the case of 2PC, every transaction undergoes two complete BFT protocol executions in each input shard, doubling the overhead compared to an intra-shard transaction. While the increased effort for cross-shard transactions is necessary to maintain consistency across shards, the associated cost is excessively high. Unfortunately, neither of the two executions is removable. The locking mechanism in 2PC involves two status updates for each available input—first from "available" to "locked", and then from "locked" to "unlocked" or "spent". Secure updating is ensured exclusively through BFT protocols.

Since the locking mechanism cannot be adopted with lower overhead, alternative approaches must be explored to reduce cross-shard transaction processing costs. RapidChain [53] introduces a *splitting* mechanism, where the output shard splits the cross-shard transaction into several "subtransactions", each processed by its respective input shard directly to spend their inputs. Monoxide [49] proposes a similar mechanism called *transaction relay*, where input shards also directly spend available inputs regardless of the transaction validity. Although the cross-shard processing overhead is decreased (i.e., a single BFT protocol execution in each involved shard), these strategies introduce severe security problems. In the case of an invalid cross-shard transaction with multiple inputs, where one input is unavailable, other available inputs are still expended. This results in the transaction being *partially executed* rather than comprehensively rejected, thereby failing to achieve atomicity.

Challenge 3: Reducing inter-shard communication overhead practically. Solving this issue might seem straightforward by processing requests with the same output shard in one intra-shard consensus, allowing the transactions collectively certified by one BFT proof. However, this mechanism is impractical. Transactions submitted to the system at any given moment vary in type and involved shards. When selecting transactions for each intra-shard consensus, prioritizing commitment for the shard with batch-sized transactions delays some earlier-arriving transactions, resulting in poor responsiveness. Alternatively, committing in chronological order for each output shard often leads to an insufficient number of transactions in each consensus to reach optimal batch size. In the worst case, when the output shards are different for each pending transaction, this method not only fails to reduce inter-shard communication costs but also seriously hampers intra-shard efficiency.

Our solutions in summary. Now we walk through how we address the challenges mentioned above and realize efficient transaction processing for sharding blockchains with robust security.

First ingredient: a paradigm modification for request submitting. We prevent malicious clients from censored submitting by putting forth a new submission paradigm. Clients submit each request to its output shard. The honest parties within the output shard scrutinize the request to ensure its structural integrity, verifying that each input is accompanied by a signature and that the output value does not surpass the total value of inputs. Well-structured requests are then transmitted to other involved shards. The submitting paradigm ensures that each involved shard reliably receives the same transaction request. Given that each shard is configured to maintain honesty, the request must be processed by each input shard. Output shards receive either all input certificates or an invalidity proof, with the confidence that the scenario of receiving neither is not possible. A malicious client has no capability to disrupt the correct processing or inflict harm upon other clients.

Second ingredient: optimal intra-shard overhead harmless on security. The reduction of intra-shard overhead fulfilled by RapidChain and other akin solutions is the right direction but it lacks security insurance for invalid transactions. Our protocol continues this effort that also commits a *x-in-y-out* cross-shard transaction through one intra-shard consensus in each involved shard. However, available inputs are not spent to the payee directly, but to the output shard "buffer". Inputs in buffer are securely managed by the shard parties with a $(n-f,n)$-threshold signature. They are finally transferred to the payee through an output shard consensus only when all inputs are received, resulting in an optimal overall $x+y$ intra-shard consensus execution for the commitment. If the output shard receives an unavailability certificate from an input shard (guaranteed by the submitting paradigm), it returns the inputs in buffer to the initial shard, thereby atomically rejecting the invalid transaction. Besides, we observe the necessary information of an unavailability certificate, and straightforwardly generate it through an intra-shard multicasting, much simpler and faster than any full-fledged BFT protocol. It results that each shard can timely acknowledge a transaction is invalid, and the processing stops immediately. Benefits from the responsive and fast unavailability certificate generation, our protocol achieves no BFT protocol execution for atomic rejection in good cases.

How to realize the buffer*?* In sharding blockchains where malicious parties, shard member reconfiguration, and intentional delays by adversaries are common challenges, it is unsafe for any single party to act as a sole receiver and independently manage these inputs. Therefore, we design an elaborate approach where all shard members collectively function as the buffer receiver. Inputs are exclusively transferred to the payee when every honest party agrees.

We implement this process using threshold cryptography. During system initialization, each shard is equipped with a $(n-f,n)$-threshold signature scheme (e.g., BLS threshold signature [9]), where the group public key serves as the buffer address. Inputs for cross-shard requests are initially spent to the output shard buffer by their respective shards, and they get committed to the transaction payee only if each honest party signs that the buffer has stored all transaction inputs. Moreover, the BFT protocols (e.g., [25, 51]) adopted in each shard naturally require threshold signature schemes for signing messages or generating public randomness. Therefore, shards can maintain buffers without incurring extra costs.

Final piece: lightweight certificate for inter-shard communication. To address the challenge of heavy inter-shard communication in cross-shard transaction processing, we introduce a novel batch mechanism that proves multiple input availability in a single certificate. As discussed earlier, processing requests with the same output shard in each consensus execution is impractical. Hence, we redirect the transaction selection and classification from occurring before intra-shard consensus to taking place after it. Each shard still picks up transactions, reaching the optimal batch size, for each BFT protocol execution in the order of their arrival. After receiving committed transaction batch, honest parties classify transactions based on their output shards and generate a certificate for each transaction set with the same output shard through a Merkle proof. This "*batch-after-consensus*" mechanism ensures that transactions are still effectively processed in chronological order while reducing inter-shard communication overhead.

3 PROBLEM FORMULATION

Byzantine fault tolerance protocol BFT**.** In the sharding blockchain system, each shard achieves intra-shard consensus through the deployment of a *Byzantine fault tolerance* protocol (denoted as BFT). BFT ensures *safety* and *liveness* despite adversaries controlling the communication network and corrupting some parties. Safety guarantees that all honest parties in the same shard eventually output the same transactions into shard ledger log, and liveness guarantees that any submitted valid transaction is eventually output to log by every honest party.

Client request req**.** Clients submit transaction requests (denoted as req) to the sharding blockchain system. A req includes client-signed transaction inputs, each within the shard it belongs to, payees' addresses (i.e., public keys), and the transaction value.

Transaction tx**.** When processing client requests, shards construct transactions denoted as tx. The format of tx is $tx = (\mathsf{type}, id, \mathrm{I}, \mathrm{O})$. type represents the type of the transaction. id denotes the transaction request ID. $\mathrm{I} = \{I_1, I_2, \cdots\}$ indicates the transaction input set, where each $I_i \in \mathrm{I}$ consists of the belonging shard S_i, unspent transaction output utxo_i, and the client's signature sig_i. $\mathrm{O} = \{O_1, O_2, \cdots\}$ denotes transaction output, where each $O_j \in \mathrm{O}$ includes the output shard S_j, payee's public key pk_j, and the output value $vals_j$.

There are three types of transactions in Kronos: Spend-Transaction (denoted as S-tx, within type = sp) for input shard spending, Finish-Transaction (denoted as F-tx, within type = fh) for output shard committing, and Back-Transaction (denoted as B-tx, within type = bk) for rolling back invalid execution.

Transaction waiting queue Q**.** Each shard maintains a *waiting queue* denoted as Q to store unprocessed transactions in the order of their arrival. During each round of BFT, a maximum of b transactions is selected from the top of Q for commitment.

Shard ledger log**.** Each shard records process completed transactions to shard ledger log with the format that $\mathsf{log} = tx_1 \parallel tx_2 \parallel \cdots$.

3.1 Cryptographic primitives

Threshold signature scheme. Let $0 \le t \le n$, a (t,n)-non interactive threshold signature scheme is a tuple of algorithms which involves n parties and up to $t-1$ parties can be corrupted. After initial key generation by function SigSetup, each node has a private function ShareSig and public functions ShareVerify, Combine and Verify. Neither the signature share nor the combined threshold signature is forgeable and the scheme is robust. See Appendix A for formal definitions.

Merkle Tree. A *Merkle tree* (or *hash tree*) uses cryptographic hashes for each "leaf" node representing a data block. The nodes higher up are hashes of their children, and the top is the tree root rt.

3.2 System and threat model

System initialization. The whole network has N nodes, which are divided into m shards. Each shard S_i (where $i = 1, 2, \cdots, m$) involves a set of parties $\{P_j\}_{j\in[n]}$, where n is the shard size and $[n]$ denotes the integers $\{1, 2, \cdots, n\}$. Each shard initializes a threshold signature scheme among the shard participants, so each party P_j can get its individual secret key sk_j and corresponding public keys. The setup can be executed through *distributed key generation* [1, 7, 16]. Notably, the group public key gpk serves as the shard buffer address receiving cross-shard inputs, so each shard's gpk is public to all participants across the network.

Network and threat model. Each party within a shard connects to each other through a reliable peer-to-peer (P2P) network. We describe the connection as *asynchronous* (which is with the weakest time assumption of all network models) in our protocol, so the generic deployment in any network is natural. In the asynchronous network, an adversary can casually delay messages or disrupt their order, but each message will be received eventually. The adversary can corrupt up to F out of N parties by taking full control of them where $N \geq 3F + 1$ (optimal fault tolerance in an asynchronous network). Additionally, there may be *malicious clients* in the system attempting to damage the system security by unconventional manners, such as refusing to sign the request, providing a fake signature, or secretly withholding messages that should be sent to some shards (as aforementioned in Section 2).

Shard configuration. We envision our protocol operating within a secure shard configuration. Each shard is considered an *honest shard*, where the number of Byzantine parties f satisfies $f < n/3$ (n is the shard size). A secure shard configuration could be realized by [3, 15, 17, 53, 54].

3.3 Security goal

We propose a secure sharding blockchain definition by incorporating security properties outlined in the notable work [6]. Our definition extends the discussion to encompass invalid transaction request processing and introduces *atomicity*. The precise definition of secure sharding blockchains is as follows:

Definition 1 (Secure sharding blockchain). *A sharding blockchain consensus operates in consecutive rounds to output committed transactions and each shard records the committed transactions to its append-only shard ledger. The protocol is secure if and only if it has a negligible probability of failing to satisfy the following properties:*

- *Persistence: If an honest party reports a transaction tx is at position k of his shard ledger in a certain round, then whenever tx is reported by any honest party it will be at the same position.*
- *Consistency: There is no round that there are two honest parties respectively report tx_1 and tx_2, where $tx_1 \neq tx_2$, in their shard ledger and tx_1 is in conflict with tx_2 (i.e. sharing the same input).*
- *Atomicity: In the context of a transaction request involving value transfer across multiple shards, all involved shards execute the required value-moving operations in their entirety during commitment (valid request), or comprehensively reject it without any final commitment or value transfer (invalid request).*
- *Liveness: Once a transaction request is submitted to the system, it is processed eventually, either executed through a committed transaction tx recorded in the shard ledger or rejected with proof.*

3.4 Performance metrics

Aiming at processing transactions for sharding blockchains at low cost, we consider the critical efficiency metrics:

- *Intra-shard communication overhead factor* (IS-COF): We primarily consider the overhead for transaction processing within each involved shard. Our protocol is plug-in to sharding blockchains that employ various BFT protocols with different communication complexities, so we analyze an overhead factor indicating the number of BFTs executed for each transaction processing. For an intra-shard one, it's straightforward to commit it through an intra-shard consensus, where the IS-COF= 1.
- *Cross/Inter-shard communication overhead factor* (CS-COF): It measures the number of messages required to be transmitted across shards for each cross-shard transaction processing.
- *Cross/Inter-shard message overhead* (CS-MO): This metric represents the (average) bits of messages transmitted across shards associated with processing each transaction.

4 KRONOS

In this section, we introduce our work, a sharding blockchain consensus realizing robust security with atomicity and optimal communication overhead in any network model.

Overview. Kronos takes clients' transaction requests as input and records transactions for valid requests in the output shard ledger. During a request processing, a client (or multiple clients) initially submits the request to its output shard (if there are multiple output shards, clients specify one for submission). After receiving complete request information, output shard disseminates the request to all involved shards. Each informed input shard processes it in an intra-shard way first. If every input managed by the current shard is available, the shard spends these inputs to the output shard through a BFT. After the transaction is committed in a BFT consensus batch, the shard notifies this expenditure to the output shard, along with other inputs spent to it through the same BFT, using a single certificate. Parties in the output shard verify the certificate and store certified inputs in "buffer" (served by the output shard's gpk). Only a threshold signature can authorize further expenditure from buffer. Once the shard accumulates all inputs of the request, it constructs a transaction to finalize the request processing. In case an input shard detects a received request containing an unavailable input, it sends a quorum proof of invalidity to other shards. Upon receiving the invalidity proof, output shards halt processing the request, respond with a rejection to the clients, and input shards abort execution and roll back if necessary.

Specific process. Figure 3 gives an example of request processing, where the request $\mathsf{req}[\alpha]$ transfers input I_1 in shard S_1 and input I_2 in S_2 to a payee in shard S_3. Each shard is equipped with a BFT that commits transactions in consecutive rounds. Similar to most blockchains, BFT commits with an external function TxVerify for transaction verification. Any transaction tx is output by BFT only if $\mathsf{TxVerify}(tx) = 1$. Figure 6 and Figure 7 illustrate how each honest party operates in a running sharding blockchain system with Kronos. The transaction request processing is as follows:

Step 1: *Output shard delivers transaction request.*

Clients initiate the process by submitting transaction request $\mathsf{req}[id]$ to the output shard S_{out}. Once $\mathsf{req}[id]$ is complete with

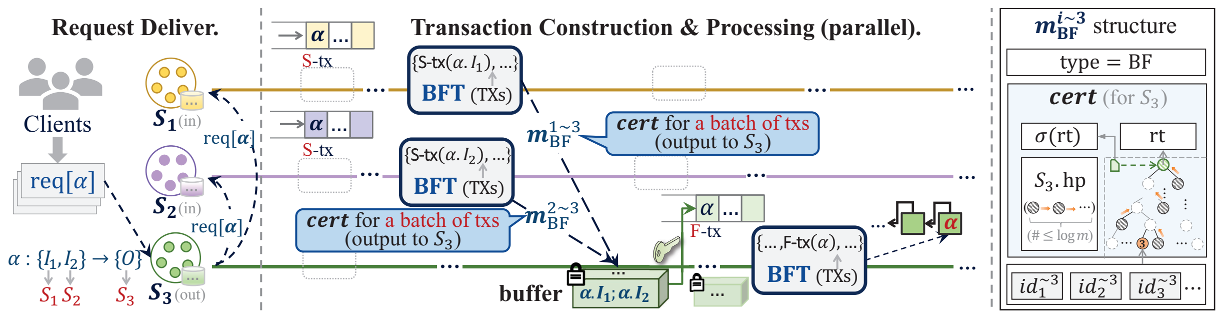

Request Deliver.
Clients
req[α]
$\alpha:\{I_1,I_2\}\rightarrow\{O\}$
S_1 S_2 S_3
S_1(in)
S_2(in)
S_3(out)
req[α]
req[α]
Transaction Construction & Processing (parallel).
S-tx
S-tx
{S-tx($\alpha.I_1$), ...}
BFT (TXs)
{S-tx($\alpha.I_2$), ...}
BFT (TXs)
$m_{BF}^{1\sim3}$
$m_{BF}^{2\sim3}$
cert for a batch of txs (output to S_3)
cert for a batch of txs (output to S_3)
buffer
$\alpha.I_1;\alpha.I_2$
F-tx
{..., F-tx(α), ...}
BFT (TXs)
$m_{BF}^{i\sim3}$ structure
type = BF
cert (for S_3)
σ(rt)
rt
S_3.hp
(# ≤ log m)
$id_1^{\sim3}$ $id_2^{\sim3}$ $id_3^{\sim3}$ ...

Figure 3: Valid transaction processing of Kronos.

all necessary information, including signatures for each input, the output shard ID, payee's public key, and ensuring the output value less than total inputs, it undergoes further process. In the case of an intra-shard request, involving transfer within S_{out} solely, the delivery process is finished. If req[id] is cross-shard, S_{out} delivers it to all other involved shards. This ensures that each involved shard receives the same request, thwarting any attempt by malicious clients to submit ambiguous requests to each shard.

Step 2: *Input shard spends available inputs with batch certification.*

After receiving req[id], honest parties in the input shard S_{in} verify whether the required inputs managed by S_{in} are available. The verification involves checking UTXO$_{\text{in}}$ and validating the client signatures. If the conditions are satisfied, they proceed to construct a Spend-Transaction, S-tx[id], for the input expenditure. If req[id] is intra-shard, the output address pk is the payee's public key. In case req[id] is cross-shard, transferring funds to other shards, the output field O is populated with the output shard ID, the output shard buffer address gpk_{out}, and the transferred value. Subsequently, S-tx[id] is added to the queue Q waiting for BFT commitment.

Transactions in Q are committed by BFT in consecutive round ℓ in order. Each round BFT$_\ell$ picks at most b transactions from the top of Q as input, and outputs committed transactions (denoted as TXs$_\ell$). For each committed S-tx[id], if the corresponding request req[id] is intra-shard, honest parties record S-tx[id] in the current shard ledger S_{in}.log, update the output to UTXO$_{\text{in}}$, and finalize the request processing by responding to the client.

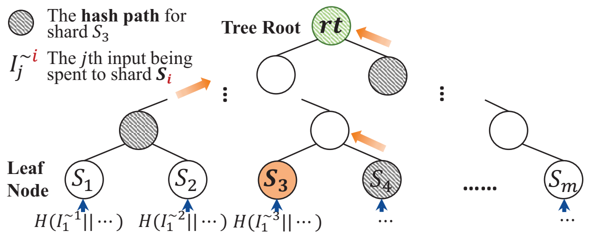

The hash path for shard S_3
$I_j^{\sim i}$ The jth input being spent to shard S_i
Tree Root
rt
Leaf Node
S_1 S_2 S_3 S_4 ...... S_m
$H(I_1^{\sim1}||\cdots)$ $H(I_1^{\sim2}||\cdots)$ $H(I_1^{\sim3}||\cdots)$

Figure 4: Shard-index tree structure.

In case req[id] is a cross-shard request, the inputs of S-tx[id] are expended to other shards, so neither S_{in}.log nor UTXO$_{\text{in}}$ needs updating. To convince the output shard of req[id] that the current input shard has spent for it, the input shard provides certification for spent inputs, relying on Merkle tree technology. In the tree construction (as shown in Figure 4), committed S-txs spending for cross-shard requests are categorized by their output shards. The inputs of S-txs sharing the same output shard are linked in lexicographic order, with the hash value serving as the leaf node of this output shard. The Merkle tree with the tree root value rt is constructed with all leaf nodes sorted in lexicographic order of the corresponding shard ID. Each leaf node has a *hash path* (denoted as hp) leading to rt, which plays an important role in the following inter-shard certification.

The Merkle tree proves multiple input expenditures to each shard with the hash path and root rt. Every honest party signs to rt, guaranteeing the tree's validity with a threshold signature σ^{rt}.

Remark: Careful observation of the tree construction reveals that the threshold signature for rt may seem redundant, as the input expenditures have already been committed by BFT whose proof can naturally prove the tree's validity. It is gratifying to note that the signing is indeed removable in synchronous or partial synchronous environments, where the adopted BFT (e.g., [51], [2], [45]) is *deterministic* and commits *with a proof* (which is usually implemented by an aggregated multi-signature, a threshold signature, or trivial signatures from $n-f$ parties). Transactions are committed by such protocols deterministically, so the tree can be constructed *before* the BFT execution. By proposing transactions along with the tree root rt in the proposal phase, the BFT proof guarantees rt validity. In asynchronous networks, however, the signing is necessary. According to the FLP "impossibility" [22], asynchronous BFT protocols (e.g., [42], [18], [26], [40], [24], [39], [25]) must run *randomized* subroutines to ensure security, leading to uncertainty about the committed transactions. Therefore, the tree can only be constructed after the BFT and then signed by the shard.

For each output shard S_{out}, current shard sends a Buffer-Message m_{BF} to inform which requests have been spent for. m_{BF} includes the request IDs and a certificate for these requests, composing tree root rt, signature of rt, and the hash path hp of S_{out}.

Step 3: *Output shard waits for integral inputs.*

When shard S_{out} receives Buffer-Message m_{BF} from S_{in}, it firstly verifies proof with S_{in} public key. According to the request IDs provided in m_{BF}, S_{out} can calculate the leaf node value of current shard. Combined with the Merkle path hp in the certificate, S_{out} can compute tree root rt′ locally. If rt′ is equal to the received rt, it is verified that all the requests in m_{BF} have been spent in S_{in}.

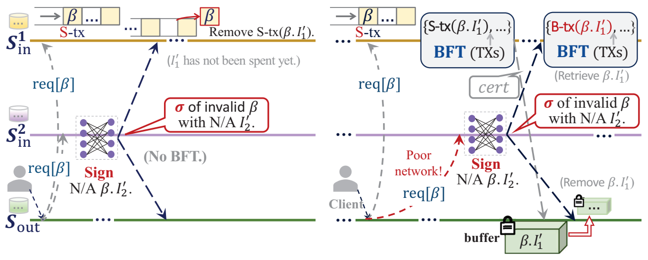
To avoid any invalid transaction's payee receives undeserved funds, S_{out} refrains from directly transferring the verified inputs to the payee addresses. Instead, it temporarily stores these inputs in buffer$_{\text{out}}$ and awaits integral inputs.

Step 4: *Output shard finalizes valid request processing.*

When an honest party in S_{out} receives all inputs of request req[id] confirming its validity, the party signs to id and multicasts the signature among S_{out} to inform that the request is ready for commitment. Once there are $n-f$ valid signatures for id, indicating all honest parties have received integral inputs, req$[id]$ inputs in buffer$_{\text{out}}$ become accessible and capable of being transferred by a Finish-Transaction, F-tx$[id]$. The input of F-tx$[id]$ comprises all inputs of req$[id]$ stored in buffer$_{\text{out}}$ along with the $(n-f, n)$-threshold signature (serving as T-SIG). When a certain round BFT outputs F-tx$[id]$, it is recorded in the shard ledger S_{out}.log and added to UTXO$_{\text{out}}$. buffer removes the stored inputs of req$[id]$, and S_{out} responses to the client to finalize the valid request processing.

Invalid transaction rejection. There are two kinds of invalid transaction requests. One is that the request is *structure-incomplete*, i.e., lacking some necessary information such as signatures of inputs or payee public keys, or the output value exceeds inputs. This kind of invalidity can be promptly identified upon submission to the output shard and rejected without further undergoing.

Another kind of invalid request is well-structured but *content-incorrect*, where the utxo is non-existent, or sig is fallacious. This incorrectness can only be verified by the input shard responsible for managing the unavailable input. Figure 5a and 5b show examples of how a content-incorrect request req$[\beta]$ is rejected in a good case or experiences a poor delay. req$[\beta]$ requests to transfer input $I_1{}'$ in shard S_1 and $I_2{}'$ in shard S_2 to shard S_3 with signatures, while $I_2{}'$ is not exists in UTXO$_2$. Since req$[\beta]$ is well-structured, the initial deliver step is operated, where S_3 delivers it to S_1 and S_2. Take req$[\beta]$ as an example, a secure and comprehensive rejection for an invalid request is achieved as follows:



(a) good case (b) worst case

Figure 5: Invalid transaction processing of Kronos.

Step 2#: *Input shard proves input unavailability requiring no* BFT.

Upon receiving req$[\beta]$, honest parties in S_2 identify that $I_2{}'$ is unavailable, so they sign a Reject-Message m_{RJ} for req$[\beta]$ and multicast it among the shard. Once req$[\beta]$ is proven invalid by $n-f$ messages with verified signatures, the invalidity is established. S_2 sends the Reject-Message to other involved shards (i.e., S_1 and S_3) with the combined $(n-f, n)$-threshold signature as proof.

Step 3#: *Output shard rejects the request.*

Upon receiving Reject-Message of req$[\beta]$, the output shard S_3 verifies the signature and subsequently removes the inputs stored for req$[\beta]$ from its buffer$_3$. Then S_3 responds to the client that req$[\beta]$ is rejected.

Step 4#: *Input shard rejects the request.*

Upon receiving req$[\beta]$, S_1 checks whether it has been processed. *Case 1 (optimistic): Abort processing.* If req$[\beta]$ has not been processed yet (as shown in Figure 5a), honest parties quit executing it and eliminate S-tx(β) from Q (if it exists). Because the Reject-Message m_{RJ} is constructed simply through an intra-shard threshold signature, the process is typically not slower than most full-fledged BFT protocols (where the round of communication is at least one in synchronous models and two or three in partial synchronous or asynchronous models). Therefore, the optimistic case often occurs with no BFT being "wasted on" invalid requests.

Case 2 (worst): Get back spent inputs. Otherwise, as Figure 5b shows, S_2 suffer a terrible latency or adversary delay, and S_1 has spent for req$[\beta]$ with S-tx$[\beta]$. For a comprehensive rejection, S_1 returns the spent input to the initial payer through a Back-Transaction B-tx$[\beta]$ with the received threshold signature as T-SIG, where the utxo is the output of the committed S-tx$[\beta]$. When B-tx$[\beta]$ is output by BFT in S_1, every honest party update UTXO with B-tx$[\beta]$.O, ensuring a comprehensive rollback.

5 SECURITY AND COMPLEXITY ANALYSIS

5.1 Security Analysis

We prove that Kronos satisfies the security properties of persistence, consistency, atomicity, and liveness indicated in Definition 1. Due to page limitations, we provide theorems that Kronos satisfies every property and their guarantees. See Appendix B for detailed proof.

Theorem 1 (Persistence). *If in a given* Kronos *round, an honest party P_i in shard S_c outputs a transaction tx at height k in shard ledger S_c.log$_i$, then tx must occupy the same position in ledger S_c.log$_j$ recorded by every honest party P_j in shard S_c.*

Proof. The persistence property relies on the majority honesty of shard configuration and safety of BFT deployed in each shard. □

Theorem 2 (Consistency). *There is no round r in which there are two honest party ledger states* log$_1$ *and* log$_2$ *with transactions tx_1, tx_2 respectively, such that $tx_1 \neq tx_2$ and tx_1.I $\cap$ tx_2.I $\neq \varnothing$.*

Proof. Consistency is ensured by TxVerify of BFT, inter-shard certification, and threshold buffer management. □

Theorem 3 (Atomicity). *A cross-shard transaction request* req$[\gamma]$ *is either executed by all involved shards, or comprehensively rejected by each shard without any fund movement if it is invalid.*

Proof. The atomicity property is ensured by the output shard waiting for integral inputs before commitment and rollback mechanism achieved with back-transactions. □

Theorem 4 (Liveness). *If a transaction request* req$[\gamma]$ *is submitted, it would undergo processing within κ rounds of communication (intra- or inter-shard), resulting in either a ledger-recorded transaction or a comprehensive rejection, where κ is the liveness parameter.*

Proof. The liveness property is guaranteed by the introduced submission paradigm and intra-shard BFT liveness. □

5.2 Complexity Analysis

We analyze the minimal intra-shard communication overhead factor for *x-in-y-out* cross-shard transaction processing and prove that our protocol achieves optimality, where IS-COF= $x+y$. Kronos ensures secure cross-shard cooperation with lightweight communication overhead factor of $x+y-1+xy$ and message complexity of $O(\frac{m\log m}{b}\lambda)$ (always below $O(\lambda)$). See Appendix C for a detailed analysis of the message overhead.

Sharding Blockchain Consensus **Kronos**

In shard S_c, Kronos proceeds as follows for each honest party P_i:

(1) **Request Deliver.**
- Upon receiving request $\mathrm{req}[id]$ submitted by clients:
  - Verify that: S_c serves as the output shard of $\mathrm{req}[id]$, each input has a signature sig, and the output value does not exceed total value of inputs. If the verification fails, consider the request invalid and ignore it.
  - If $\mathrm{req}[id]$ is a verified cross-shard request, send $\mathrm{req}[id]$ to other involved shards.

(2) **Transaction Construction.**
- Spend-Transaction (S-tx): Upon receiving $\mathrm{req}[id]$:
  - Verify that: each input I managed by current shard S_c holds $I.\mathrm{utxo} \in \mathrm{UTXO}_c$, and $I.\mathrm{sig}$ is valid. *// Verify the availability.*
  - If some I' verification fails and $\mathrm{req}[id]$ is cross-shard, invoke $\mathbf{RejCSReq}(id, I')$. *// To reject the invalid request comprehensively.*
  - Else, construct $\text{S-tx}[id] = (\textsc{sp}, id, \mathrm{I}, \mathrm{O})$, where I is set of Is to be spent.
    - If $\mathrm{req}[id]$ is intra-shard, set $\mathrm{O} = (S_c, pk, vals)$ where pk is the public key of payee.
    - Otherwise, $\mathrm{O} = (S_{\mathrm{out}}, gpk_{\mathrm{out}}, \mathrm{I}.vals)$ where gpk_{out} is the public key of output shard buffer. *// S_c is an input shard of* $\mathrm{req}[id]$.
- Finish-Transaction (F-tx): If every I of $\mathrm{req}[id]$ has been stored in buffer: *// S_c is the output shard of valid cross-shard* $\mathrm{req}[id]$.
  - Sign $s_i{}^{\mathrm{FH}} := \mathsf{ShareSig}(sk_i, \mathsf{H}(\{I\}))$ and multicast $(id, s_i{}^{\mathrm{FH}})$ among S_c. *// Apply to execute* $\mathrm{req}[id]$.
  - Upon receiving $n-f$ valid (id, s_j) from distinct parties P_j that $\mathsf{ShareVerify}(\mathsf{H}(\{I\}), \{j, s_j\}) = 1$:
    - Compute $\sigma^{\mathrm{FH}}[id] := \mathsf{Combine}(\mathsf{H}(\{I\}), \{j, s_j\}_{n-f})$. *// Serve as the validity proof of* F-tx$[id]$.
    - Construct $\text{F-tx}[id] = (\textsc{fh}, id, \mathrm{I}, \mathrm{O})$, where $\mathrm{I} = \langle \{I\}, \sigma^{\mathrm{FH}}[id] \rangle$ and $\mathrm{O} = (S_c, pk_c, vals_c)$. *// Transfer received inputs to the payee.*
- Back-Transaction (B-tx): Upon receiving $\langle m_{\mathrm{RJ}}[id], \sigma^{\mathrm{RJ}} \rangle)$ from shard S' where $\mathsf{Verify}(m_{\mathrm{RJ}}[id], \sigma^{\mathrm{RJ}}) = 1$ and $\text{S-tx}[id]$ has been output by a certain BFT round in S_c: *//* $\mathrm{req}[id]$ *is invalid, but current shard S_c has spent for it.*
  - Construct $\text{B-tx} = (\textsc{bk}, id, \mathrm{I}, \mathrm{O})$, where $\mathrm{I} = \text{S-tx}[id].\mathrm{O}$ with $\text{T-SIG} = \sigma^{\mathrm{RJ}}$. $\mathrm{O} = (S_c, pk, vals)$ where pk is the initial address of the spent I and $vals$ is its value. *// Get back the misspent inputs.*

After construction, each transaction is appended to the waiting queue Q.

. . . .

Every shard S_c processes transactions in Q by BFT_ℓ in consecutive round number ℓ. Before each round BFT_ℓ, S_c initials the set of S-txs for cross-shard requests with $\mathrm{cTXs}_\ell = \varnothing$. The honest party P_i in S_c operates as follows: (Note that operations indexed with "$\circ$" can be *omitted* if BFT already possesses an existing proof. Refer to Step 2 for more details.)

(3) **Intra-Shard Consensus & State Update.**
- Execute BFT_ℓ to commit transactions at the top of Q with the batch size b.
- Upon receiving committed TXs_ℓ from BFT_ℓ, for each $tx_i \in \mathrm{TXs}_\ell$:
  - If $tx_i.\mathrm{type} = \textsc{sp}$: *// tx_i spends inputs for requests.*
    - For each $I \in tx_i.\mathrm{I}$: If $I.\mathrm{utxo} \in \mathrm{UTXO}$, $\mathrm{UTXO} \leftarrow \mathrm{UTXO} \backslash I.\mathrm{utxo}$. *// Remove the spent* utxo.
    - If $tx_i.\mathrm{O} = (S_c, \cdot, \cdot)$, update $\log \leftarrow \log \,\|\, tx_i$ and $\mathrm{UTXO} \leftarrow \mathrm{UTXO} \cup tx_i.\mathrm{O}$. *// tx_i commits an intra-shard request.*
    - Otherwise, set $\mathrm{cTXs}_\ell \leftarrow \mathrm{cTXs}_\ell \cup tx_i$. *// tx_i spends inputs for a cross-shard request.*
  - If $tx_i.\mathrm{type} = \textsc{fh}$, update $\log \leftarrow \log \,\|\, tx_i$ and $\mathrm{UTXO} \leftarrow \mathrm{UTXO} \cup tx_i.\mathrm{O}$. *// Finalize the cross-shard request processing.*
  - If $tx_i.\mathrm{type} = \textsc{bk}$, update $\mathrm{UTXO} \leftarrow \mathrm{UTXO} \cup tx_i.\mathrm{O}$. *// Get back the* utxo *spent for some invalid request.*
- If $\mathrm{cTXs}_\ell \neq \varnothing$: *// Some inputs are spent to other output shards by* BFT_ℓ*, requiring the provision of certificates for them.*
  - Connstruct the shard-index tree with cTXs_ℓ. *// Construct Merkle tree for batch certification.*

  // Non-essential operations indexed with "$\circ$":
  - $\circ$ Sign $s_i{}^{\mathrm{rt}} := \mathsf{ShareSig}(sk_i, \mathrm{rt}_\ell))$ and multicast $(\ell, s_i{}^{\mathrm{rt}})$ among S_c.
  - $\circ$ Upon receiving $n-f$ valid (ℓ, s_j) from distinct parties P_j that $\mathsf{ShareVerify}(\mathrm{rt}_\ell, \{j, s_j\}) = 1$:
    - Compute $\sigma^{\mathrm{rt}}[\ell] := \mathsf{Combine}(\mathrm{rt}_\ell, \{j, s_j\}_{n-f})$ as proof_ℓ. *// Prove that* (tree, rt) *is valid.*
  - Execute operations in (4).
- Upon receiving $\langle m_{\mathrm{RJ}}[id], \sigma^{\mathrm{RJ}} \rangle$ from shard S' where $\mathsf{Verify}(m_{\mathrm{RJ}}[id], \sigma^{\mathrm{RJ}}) = 1$ and S_c is the output shard of $\mathrm{req}[id]$:
  - $\mathrm{buffer} := \mathrm{buffer} \backslash \{id, \mathrm{I}\}$. *// Remove the inputs stored for invalid* $\mathrm{req}[id]$ *from* buffer.

(4) **Inter-Shard Transaction Batch Certification.** *// Required only when* BFT_ℓ *commits spending some inputs to other shards.*
- For each shard S' with a leaf node in tree_ℓ: *// S' is the output shard of some* S-txs.
  - Construct Buffer-Message $m^\ell_{\mathrm{BF}}(S') = (\textsc{bf}, \{id\}, \mathrm{cert}'_\ell)$ where $\mathrm{cert}'_\ell = (\mathrm{proof}_\ell, \mathrm{rt}_\ell, S'.\mathrm{hp}_\ell)$, and send $m^\ell_{\mathrm{BF}}(S')$ to S'.
- Upon receiving $m_{\mathrm{BF}}(S_c) = (\textsc{bf}, \{id\}, \mathrm{cert})$ from some shard S'' and $\mathsf{BufVerify}(m_{\mathrm{BF}}, S'') = 1$:
  - $\mathrm{buffer} \leftarrow \mathrm{buffer} \cup \{I\}$ where each $I = (S'', \cdot, \cdot)$ is certified being spent on a certain $\mathrm{req}[id]$ with $id \in \{id\}$.

Internal Function $\mathbf{RejCSReq}(\boldsymbol{id}, \boldsymbol{I'})$ // Provide an invalidity proof to comprehensively reject an invalid cross-shard request.
- Construct Reject-Message $m_{\mathrm{RJ}}[id] = (\textsc{rj}, id, I')$ where I' is the unavailable input.
- Sign $s_i{}^{\mathrm{RJ}} := \mathsf{ShareSig}(sk_i, m_{\mathrm{RJ}}[id])$ and multicast $\langle m_{\mathrm{RJ}}[id], s_i{}^{\mathrm{RJ}} \rangle$ among S_c.
- Upon receiving $n-f$ valid $\langle m_{\mathrm{RJ}}[id], s_j \rangle$ from distinct parties P_j that $\mathsf{ShareVerify}(m_{\mathrm{RJ}}[id], (j, s_j)) = 1$:
  - Compute $\sigma^{\mathrm{RJ}}[id] := \mathsf{Combine}(m_{\mathrm{RJ}}[id], \{(j, s_j)\}_{n-f})$ and send $\langle m_{\mathrm{RJ}}[id], \sigma^{\mathrm{RJ}}[id] \rangle$ to every involved shard of $\mathrm{req}[id]$.

Figure 6: Sharding blockchain consensus Kronos

```
External Function TxVerify($tx$) → 0/1 // The transaction verification function.
• Parse $tx$ = (type, $id$, I, O).
• If type = SP: // I consists of one or several input Is, each with a client signature $\mathsf{sig}_i$.
  – If for each $I_i \in$ I where $I_i = \langle S_c, \mathsf{utxo}_i, \mathsf{sig}_i \rangle$, $\mathsf{utxo}_i \in$ UTXO and $\mathsf{sig}_i$ is verified with $\mathsf{utxo}_i.pk$, return 1;
  – Otherwise, return 0 and revoke RejCSReq($id, I'$) where the verification of $I'$ has failed.
• If type = FH: // I includes $I_i$s with a combined signature T-SIG = $\sigma^{FH}$ of current shard $S_c$.
  – If Verify(H($\{I_i\}$), $\sigma^{\mathrm{FH}}$) = 1 where I = ($\{I_i\}$, $\sigma^{\mathrm{FH}}$), return 1; Otherwise, return 0.
• If type = BK: // $tx$ is constructed upon receiving a reject-message $m_{RJ}$ with a signature $\sigma^{RJ}$ from some shard.
  – If Verify($m_{\mathrm{RJ}}, \sigma^{\mathrm{RJ}}$) = 1 where I.T-SIG = $\sigma^{\mathrm{RJ}}$, return 1; Otherwise, return 0.

External Function BufVerify($m_{\mathrm{BF}}, S$) → 0/1 // The BUFFER-MESSAGE verification function.
• Parse $m_{\mathrm{BF}}$ = (BF, $\{id\}$, cert) where cert = (proof, rt, hp).
• Verify whether Verify(rt, proof) = 1. If not, return 0.
  // If proof is served by a BFT proof rather than a threshold signature, this verification is replaced with corresponding proof verification.
• Compute leaf′ ← H($\{I\}$) where each $I$ is going to be spend on req[$id$] by shard $S$ for each $id \in \{id\}$, and rt′ ← (leaf′, hp).
• If rt′ = rt, return 1; Otherwise, return 0.
```

Figure 7: External functions

THEOREM 5 (MINIMAL INTRA-SHARD COMMUNICATION OVERHEAD FACTOR). Kronos *commits a x-in-y-out transaction tx through executing BFT protocols $x + y$ times totally. The intra-shard communication overhead factor IS-COF= $x + y$, which is optimal.*

PROOF. A cross-shard transaction is executed with a state update to the UTXO or shard ledger log in each involved shard. Secure updating is ensured only through a BFT consensus, requiring at least one consensus in each shard. If a transaction tx is committed in fewer protocol rounds than the total shard number, specifically $x + y - 1$ rounds, there must be an involved shard $\widehat{S}$ that fails to achieve consensus on tx. Since tx is committed, $\widehat{S}$ cannot be an output shard. This implies that $\widehat{S}$ must be an input shard that does not commit to spending input $\widehat{I}$ to tx. The process does not satisfy atomicity as tx is not processed consistently by each involved shard (some commit while others do not). Additionally, $\widehat{I}$ can be spent in another transaction, resulting in double-spending. Therefore, $x + y$ is the lower bound of IS-COF for valid transaction processing, and any reduction is considered insecure.

In Kronos, input shards spend inputs, and output shards commit transactions, each through a round of BFT protocol, where the IS-COF is equal to the lowest bound $x + y$. Kronos processes invalid cross-shard transactions atomically with minimal cost, too. Invalid transactions never occupy the BFT protocol workload of their output shards because no availability certificate for unavailable inputs can be received. Invalid transactions also do not occupy the BFT protocol workload of input shards managing unavailable inputs, as a quorum proof for rejection is sufficient. Other input shards quit the transaction processing once they receive the valid proof. In this case, Kronos processes invalid transactions with the optimal overhead without requiring any BFT protocol execution.

To ensure atomicity, Kronos allows rollback by input shards that have already spent for an invalid transaction request. The inputs are returned through another round of BFT protocol, leading to 2 rounds of BFT inside the shard for rejection. Consider an invalid transaction with k unavailable inputs, the worst-case IS-COF for it is $2(x - k)$. In Kronos, this worst situation rarely occurs due to responsive rejection. Any available input gets spent only after the S-tx reaches the top b of the waiting queue Q and is then committed through a full-fledged BFT protocol round. On the other hand, the message for rejection is constructed responsively once receiving the invalid request and only requires signatures from $n - f$ parties of the shard, making it faster than the spending process. Therefore, Kronos achieves optimistic IS-COF requiring no BFT for invalid transactions in most instances, and there are at most $2(x - k)$ rounds of BFT protocol executed for it without any extra storage or computation overhead even in the worst case. □

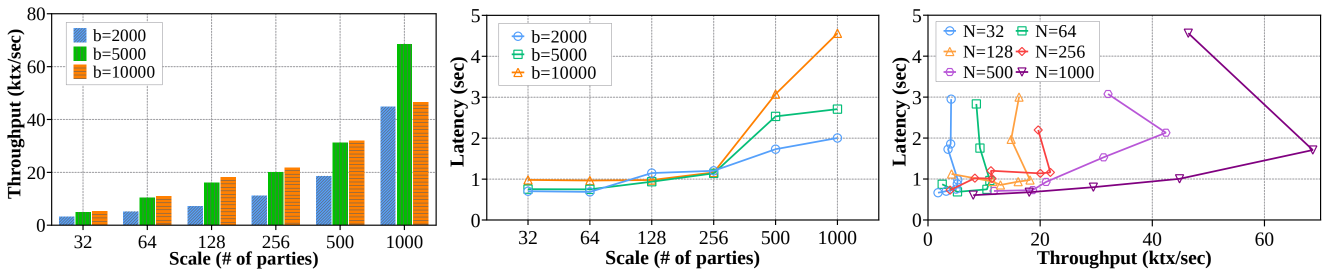
6 EVALUATION

We implement a prototype of Kronos and deploy it across 4 AWS regions (Virginia, Hong Kong, Tokyo, and London), involving up to 1000 nodes, to evaluate the practical performance. The primary aspects we want to evaluate include the overall performance of Kronos in realistic wide-area network (Section 6.1), the performance improvement compared to existing sharding protocols (Section 6.2), and whether it is truly generic and scalable in various network models with different BFT protocols (Section 6.3).

Implementation details. We program the implementations of Kronos and 2PC in the same language (i.e., Python). All libraries and security parameters required in cryptographic implementations are the same. All nodes are assigned into shards, and each shard adopts Speeding Dumbo [25] (an efficient and robust BFT protocol in asynchronous environments) with an ECDSA signature for quorum proofs and buffer management. To demonstrate the generality of Kronos, we also replace Speeding Dumbo with a well-performed partial synchronous protocol, HotStuff [51]. All hash functions are instantiated using SHA256 [31].

For notations, sKronos denotes Kronos using Speeding Dumbo for intra-shard consensus, hKronos denotes the other instantiation using HotStuff, and s2PC represents 2PC using Speeding Dumbo.

Setup on Amazon EC2. We implement sKronos, s2PC, and hKronos among Amazon EC2 c5.4xlarge instances which are



(a) Peak Throughput (b) Latency (c) Latency-Throughput Trade-off

Figure 8: Performance of sKronos **in the WAN setting.**

equipped with 16 vCPUs and 32GB main memory. The performances are evaluated with varying scales at $N = 32, 64, 128, 256, 500$, and 1000 nodes. Each EC2 instance is shared by 8 to 16 nodes. The proportion of cross-shard transactions varies between 10%, 50%, and 90%. Each cross-shard transaction involves 2 input shards and 1 output shard randomly. The transaction length is 250 bytes, which approximates the size of basic Bitcoin transactions.

6.1 Overall performance of Kronos

Throughput and Latency. To evaluate the impact of Kronos on sharding blockchains, we measure throughput, expressed as the number of requests processed per second. We vary the network size from $N = 32$ to 1000 nodes and adjust batch sizes of intra-shard consensus (i.e., the number of proposed transactions) from $b = 1$k to 30k for evaluation. This reflects how well Kronos performs in a sharding blockchain system deployed in realistic scenarios.

As illustrated in Figure 8a, sKronos demonstrates scalability, showcasing an increasing throughput as the network scales and achieving a peak throughput of 68.6ktx/sec with $N = 1000$ and $b = 5$k. For medium-sized networks ($N \leq 256$), throughput continues to grow until the batch size reaches 10k. The throughput decrease with an enlarged batch size in large-scale networks ($N = 1000$) is not indicative of a scalability loss but rather results from limitations in bandwidth or computing resources. The scalability of sKronos remains unaffected, provided that the batch size is carefully optimized.

Figure 8b illustrates sKronos latency across varying network sizes, where latency is defined as the time elapsed between the moment a request enters the waiting queue and the moment the request processing is completed. The latency is limited to a maximum of 0.97sec for network scales $N \leq 128$ and batch sizes $b \leq 10$k. With latency consistently below 1.8sec with the apposite batch size, our protocol demonstrates its effectiveness for latency-critical applications even at a large scale, such as $N = 1000$.

Throughput-latency trade-off. Figure 8c illustrates the latency-throughput trade-off of sKronos. The latency stays below 1sec, with throughput reaching 5.31ktx/sec in a small-scale network ($N = 32$) and 20.03ktx/sec in a medium-scale network ($N = 128$). In large-scale networks ($N = 500$ and 1000), the latency at peak throughput remains below 2.14sec. This trade-off underscores the applicability of Kronos in scenarios requiring both throughput and latency.

6.2 Performance on cooperation across shards and comparison with existing solutions

To analyze how well Kronos handles cross-shard requests, we evaluate the specific time cost of each processing step of sKronos, and compare the throughput and latency of sKronos and s2PC with varying cross-shard transaction proportions.

Cross-Shard Latency. It is critical to understand the specific cost of cross-shard request processing in Kronos. We evaluate this overhead by measuring the latency at each step during a cross-shard request processing, as shown in Figure 9a. Once delivered to every involved shard, a valid cross-shard request undergoes three distinct steps: input shard spending, cross-shard certification, and output shard buffer committing. We focus on the cross-shard time cost (denoted as "CS-latency"), while the time for the other two steps approximates that of the deployed BFT protocol.

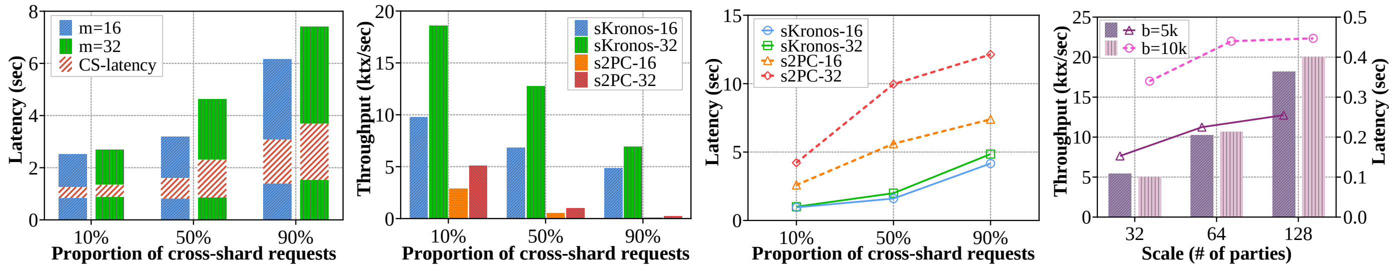
The experimental results reveal the cost of inter-shard cooperation highlighted in red (with the same batch size $b = 10$k). When the shard number m is 16 and the cross-shard request proportion is 10%, the inter-shard time cost is about 0.43sec, occupying less than 17% of the total latency. As the cross-shard request proportion increases to 90%, the impact on latency slightly rises but stays below 28% of the total latency. In a larger-scale system with 32 shards, the impact remains approximately 17% with 10% cross-shard requests and increases to no more than 30% when the cross-shard proportion is elevated to 90%. This illustrates Kronos adaptability to systems with a high frequency of cross-shard requests.

Comparison with 2PC. Furthermore, we compare the performance of Kronos with the existing cross-shard transaction processing mechanism.

Figure 9b and Figure 9c depict the throughput and latency of sKronos in comparison to s2PC. Overall, sKronos outperforms s2PC in all cases. Notably, the throughput of sKronos exceeds 3× that of s2PC when 10% of requests are cross-shard, approximately 13× when the proportion is 50%, and an impressive 42× when the proportion is 90%! In terms of latency, sKronos incurs at most half the time cost of s2PC (when the cross-shard proportion is 90%).

6.3 Performance on Various BFT

Finally, we substitute the intra-shard consensus with HotStuff to showcase Kronos generality across different systems. As Figure 9d shows, hKronos exhibits higher efficiency without complicated



(a) sKronos Cross-shard Latency (b) sKronos v.s. s2PC (TPS) (c) sKronos v.s. s2PC (Latency) (d) hKronos Performance

Figure 9: The efficiency and generality of Kronos **(**sKronos **and** s2PC **utilize** Speeding Dumbo**, and** hKronos **utilizes** HotStuff**).**

randomness in Speeding Dumbo, achieving a peak throughput of 20.2ktx/sec with a low latency of 0.45sec when $N = 128$.

Figure 10 illustrates latency-throughput trade-offs of sKronos, hKronos, and s2PC. sKronos reaches at least 3× higher peak throughput than s2PC while maintaining latency around 1sec. hKronos throughput is much higher than s2PC by an order. These results indicate that Kronos is generic in any network environment with various BFT protocols for enhancing blockchain scalability.

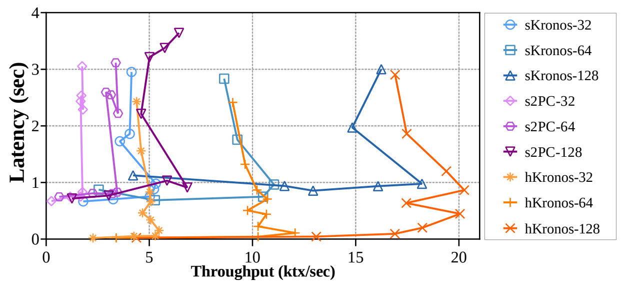


Figure 10: Latency v.s. Throughput of Several Consensus.

7 DISCUSSION AND FUTURE WORK

Replaceable certificate generation. To ensure compatibility with a variety of sharding blockchains, Kronos generates cross-shard transaction batch certificates using Merkle tree technology, which is prevalently used in blockchain systems to organize transactions and abstract block headers. Therefore, Kronos achieves lightweight inter-shard cooperation without other additional components. While Kronos also supports the integration of more efficient alternatives, such as *KZG polynomial commitment* [8, 32]. After receiving committed transactions from BFT, the shard constructs a polynomial by encoding each transaction set with the same output shard as a coefficient and generates a KZG commitment for this polynomial with threshold $n - f$. The inter-shard certificate consists of the KZG commitment and the points used for verification, allowing the output shard to verify the input expenditures with lower message complexity and verification costs. We consider this work as one of our future research directions.

Inter-shard communication paradigm. Inter-shard communication is also crucial for ensuring sharding blockchain security. It is essential that each cross-shard message can reliably reach all honest parties in the destination shard to facilitate cross-shard transaction processing. Inappropriate transmission, such as sending messages only to the shard leader, may lead to system failures under poor networks or malicious leaders. A straightforward strategy is "*all-to-all*"/"$f + 1$*-to-*$f + 1$" (adopted in [3]), which ensures security but comes with high costs. Various works propose efficient methods, including *gossip* [47, 53], *erasure code* (EC) [50], *verifiable information disperse* (VID) [5, 11], *reliable broadcast* (RBC) [12]. Kronos does not prescribe specific inter-shard communication methods. Any paradigm that ensures message secure arrival is feasible.

Secure management of buffer **during shard reconfiguration.** In the context of managing buffer, there is a minor consideration related to shard reconfiguration. To safeguard against the potential threat of a specific shard falling under complete control due to a gradual corruption of more than 1/3 of its parties, leading to a compromise of the overall system security, timely reconfiguration of shards with member changes is essential. When a shard reconfiguration happens while some inputs are still in buffer (i.e., the transaction processing has not finished yet), new shard members must "take over" buffer by acquiring comprehensive information about each input. This allows them to transfer or refund the inputs to complete the transaction processing. A DKG among the new shard members may lead to a new threshold public key, necessitating an update to the address of buffer. Each shard multicasts the updated group public key (i.e., the new address of buffer) for future input receiving, and old shard members transfer inputs stored in the old buffer to the new address through a "checkpoint" mechanism.

This process can be streamlined with the implementation of *dynamic-committee proactive secret sharing* (DPSS) [48, 52]. DPSS shares a secret among a committee and refreshes the secret shares during committee reconfiguration *without changing or revealing the original secret*, and old shares are invalid. Integrating DPSS into the DKG process during shard reconfiguration ensures a fixed share public key and buffer address, and old shard members (might be corrupted) cannot execute any operations on the buffer.

8 CONCLUSION

We present Kronos, the first *generic* sharding blockchain consensus that realizes robust security even in asynchronous networks and optimal communication overhead. Based on a leveraged buffer & batch mechanism, Kronos ensures security and efficiency without relying on any client honesty or time assumptions. Implementation results demonstrate Kronos outstanding scalability, surpassing existing solutions in all cases, making it suitable for throughput-critical and latency-critical applications. Several intriguing questions remain: theoretically, exploring more efficient methods for generating batch certificates is worth investigating; also, Kronos could be further generalized using other latest BFT protocols, such as FIN (CCS'23) [19] and ParBFT (CCS'23) [14].

REFERENCES

[1] Ittai Abraham, Philipp Jovanovic, Mary Maller, et al. 2021. Reaching consensus for asynchronous distributed key generation. In *PODC'21*. 363–373.

[2] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. 2020. Sync hotstuff: Simple and practical synchronous state machine replication. In *SP'20*. IEEE, 106–118.

[3] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, et al. 2018. Chainspace: A sharded smart contracts platform. In *NDSS'18*. ISOC.

[4] Faten Adel Alabdulwahhab. 2018. Web 3.0: the decentralized web blockchain networks and protocol innovation. In *ICCAIS'18*. IEEE, 1–4.

[5] Nicolas Alhaddad, Sourav Das, Sisi Duan, et al. 2022. Asynchronous Verifiable Information Dispersal with Near-Optimal Communication. *Cryptology ePrint Archive* (2022).

[6] Georgia Avarikioti, Antoine Desjardins, Eleftherios Kokoris-Kogias, and Roger Wattenhofer. 2023. Divide and Scale: Formalization and Roadmap to Robust Sharding. arXiv:1910.10434 [cs.DC]

[7] Fabrice Benhamouda, Shai Halevi, Hugo Krawczyk, et al. 2022. Threshold Cryptography as a Service (in the Multiserver and YOSO Models). In *CCS'22*. ACM, 323–336.

[8] Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. 2020. Efficient polynomial commitment schemes for multiple points and polynomials. *Cryptology ePrint Archive* (2020).

[9] Dan Boneh, Ben Lynn, and Hovav Shacham. 2004. Short signatures from the Weil pairing. *Journal of cryptology* 17 (2004), 297–319.

[10] Vitalik Buterin. 2017. Ethereum sharding faq. https://vitalik.ca/general/2017/12/31/sharding_faq.html.

[11] Christian Cachin and Stefano Tessaro. 2005. Asynchronous verifiable information dispersal. In *SRDS'05*. IEEE, 191–201.

[12] Jo-Mei Chang and Nicholas F. Maxemchuk. 1984. Reliable broadcast protocols. *TOCS* 2, 3 (1984), 251–273.

[13] Zhihua Cui, XUE Fei, Shiqiang Zhang, Xingjuan Cai, Yang Cao, Wensheng Zhang, and Jinjun Chen. 2020. A hybrid blockchain-based identity authentication scheme for multi-WSN. *IEEE Trans Serv Comput.* 13, 2 (2020), 241–251.

[14] Xiaohai Dai, Bolin Zhang, Hai Jin, and Ling Ren. 2023. ParBFT: Faster Asynchronous BFT Consensus with a Parallel Optimistic Path. In *CCS'23*. 504–518.

[15] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, et al. 2019. Towards scaling blockchain systems via sharding. In *SIGMOD'19*. ACM, 123–140.

[16] Sourav Das, Thomas Yurek, Zhuolun Xiang, et al. 2022. Practical asynchronous distributed key generation. In *SP'22*. IEEE, 2518–2534.

[17] Bernardo David, Bernardo Magri, Christian Matt, et al. 2022. GearBox: Optimal-size Shard Committees by Leveraging the Safety-Liveness Dichotomy. In *CCS'22*. 683–696.

[18] Sisi Duan, Michael K Reiter, and Haibin Zhang. 2018. BEAT: Asynchronous BFT made practical. In *CCS'18*. ACM, 2028–2041.

[19] Sisi Duan, Xin Wang, and Haibin Zhang. 2023. Fin: Practical signature-free asynchronous common subset in constant time. In *CCS'23*. 815–829.

[20] Christian Esposito, Massimo Ficco, and Brij Bhooshan Gupta. 2021. Blockchain-based authentication and authorization for smart city applications. *Information Processing & Management* 58, 2 (2021), 102468.

[21] Lei Feng, Yiqi Zhao, Shaoyong Guo, et al. 2021. BAFL: A blockchain-based asynchronous federated learning framework. *IEEE Trans. Comput.* 71, 5 (2021), 1092–1103.

[22] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. 1985. Impossibility of distributed consensus with one faulty process. *JACM* 32, 2 (1985), 374–382.

[23] Matthias Fitzi, Peter Ga, Aggelos Kiayias, and Alexander Russell. 2018. Parallel chains: Improving throughput and latency of blockchain protocols via parallel composition. https://eprint.iacr.org/2018/1119.pdf.

[24] Yingzi Gao, Yuan Lu, Zhenliang Lu, et al. 2022. Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency. In *CCS'22*. ACM, 1187–1201.

[25] Bingyong Guo, Yuan Lu, Zhenliang Lu, et al. 2022. Speeding Dumbo: Pushing Asynchronous BFT Closer to Practice. (2022).

[26] Bingyong Guo, Zhenliang Lu, Qiang Tang, et al. 2020. Dumbo: Faster asynchronous bft protocols. In *CCS'20*. ACM, 803–818.

[27] Zicong Hong, Song Guo, and Peng Li. 2022. Scaling blockchain via layered sharding. *IEEE J. Sel. Areas Commun.* 40, 12 (2022), 3575–3588.

[28] Zicong Hong, Song Guo, Peng Li, and Wuhui Chen. 2021. Pyramid: A layered sharding blockchain system. In *INFOCOM'21*. IEEE, 1–10.

[29] Huawei Huang, Xiaowen Peng, Jianzhou Zhan, et al. 2022. Brokerchain: A cross-shard blockchain protocol for account/balance-based state sharding. In *INFOCOM'22*. IEEE, 1968–1977.

[30] Thien Huynh-The, Thippa Reddy Gadekallu, Weizheng Wang, et al. 2023. Blockchain for the metaverse: A Review. *Futur. Gener. Comp. Syst.* (2023).

[31] Alok Kumar Kasgar, Jitendra Agrawal, and Satntosh Shahu. 2012. New modified 256-bit md 5 algorithm with sha compression function. *Int. J. Comput. Appl. Technol.* 42, 12 (2012).

[32] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. 2010. Constant-size commitments to polynomials and their applications. In *ASIACRYPT'10*. Springer, 177–194.

[33] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, et al. 2018. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *SP'18*. IEEE, 583–598.

[34] Yuzheng Li, Chuan Chen, Nan Liu, et al. 2020. A blockchain-based decentralized federated learning framework with committee consensus. *IEEE Network* 35, 1 (2020), 234–241.

[35] Yijing Lin, Zhipeng Gao, Hongyang Du, et al. 2023. A unified blockchain-semantic framework for wireless edge intelligence enabled web 3.0. *IEEE Wirel. Commun.* (2023).

[36] Yizhong Liu, Jianwei Liu, Marcos Antonio Vaz Salles, et al. 2022. Building blocks of sharding blockchain systems: Concepts, approaches, and open problems. *Comput. Sci. Rev.* 46 (2022), 100513.

[37] Yizhong Liu, Jianwei Liu, Qianhong Wu, et al. 2020. SSHC: A secure and scalable hybrid consensus protocol for sharding blockchains with a formal security framework. *IEEE Trans. Dependable Secur. Comput.* 19, 3 (2020), 2070–2088.

[38] Yizhong Liu, Xinxin Xing, Haosu Cheng, et al. 2023. A Flexible Sharding Blockchain Protocol Based on Cross-Shard Byzantine Fault Tolerance. *IEEE Trans. Inf. Forensics Secur.* 18 (2023), 2276–2291.

[39] Yuan Lu, Zhenliang Lu, and Qiang Tang. 2022. Bolt-dumbo transformer: Asynchronous consensus as fast as the pipelined bft. In *CCS'22*. ACM, 2159–2173.

[40] Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. 2020. Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited. In *PODC'20*. ACM, 129–138.

[41] Loi Luu, Viswesh Narayanan, Chaodong Zheng, et al. 2016. A secure sharding protocol for open blockchains. In *CCS'16*. ACM, 17–30.

[42] Andrew Miller, Yu Xia, Kyle Croman, et al. 2016. The honey badger of BFT protocols. In *CCS'16*. ACM, 31–42.

[43] Dimitris Mourtzis, John Angelopoulos, and Nikos Panopoulos. 2023. Blockchain integration in the era of industrial metaverse. *Applied Sciences* 13, 3 (2023), 1353.

[44] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review* (2008), 21260.

[45] Ray Neiheiser, Miguel Matos, and Luís E. T. Rodrigues. 2021. Kauri: Scalable BFT Consensus with Pipelined Tree-Based Dissemination and Aggregation. In *SOSP'21*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 35–48.

[46] Muhammad Shayan, Clement Fung, Chris JM Yoon, and Ivan Beschastnikh. 2020. Biscotti: A blockchain system for private and secure federated learning. *IEEE Trans. Parallel Distrib. Syst.* 32, 7 (2020), 1513–1525.

[47] Georgios Tsimos, Julian Loss, and Charalampos Papamanthou. 2022. Gossiping for communication-efficient broadcast. In *CRYPTO'22*. Springer, 439–469.

[48] Robin Vassantlal, Eduardo Alchieri, Bernardo Ferreira, and Alysson Bessani. 2022. Cobra: Dynamic proactive secret sharing for confidential bft services. In *SP'22*. IEEE, 1335–1353.

[49] Jiaping Wang and Hao Wang. 2019. Monoxide: Scale out Blockchains with Asynchronous Consensus Zones. In *NSDI'19*, Vol. 2019. 95–112.

[50] Stephen B Wicker and Vijay K Bhargava. 1999. *Reed-Solomon codes and their applications*. John Wiley & Sons.

[51] Maofan Yin, Dahlia Malkhi, Michael K Reiter, et al. 2019. HotStuff: BFT consensus with linearity and responsiveness. In *PODC'19*. ACM, 347–356.

[52] Thomas Yurek, Zhuolun Xiang, Yu Xia, and Andrew Miller. 2023. Long Live The Honey Badger: Robust Asynchronous {DPSS} and its Applications. In *USENIX Security'23*. USENIX Association, 5413–5430.

[53] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. 2018. Rapidchain: Scaling blockchain via full sharding. In *CCS'18*. ACM, 931–948.

[54] Mengqian Zhang, Jichen Li, Zhaohua Chen, et al. 2022. An efficient and robust committee structure for sharding blockchain. *IEEE Trans. Cloud Comput.* (2022).

[55] Peilin Zheng, Quanqing Xu, Zibin Zheng, et al. 2022. Meepo: Multiple Execution Environments per Organization in Sharded Consortium Blockchain. *IEEE J. Sel. Areas Commun.* 40, 12 (2022), 3562–3574.

A DETAILED CRYPTOGRAPHIC COMPONENTS

Threshold signature scheme. Let $0 \leq t \leq n$, a (t, n)-non interactive threshold signature scheme is a tuple of algorithms which involves n parties and up to $t-1$ parties can be corrupted. The threshold signature scheme has the following algorithms:

- *Key Generation Algorithm:* $\mathsf{SigSetup}(1^\lambda, n, t) \rightarrow \{gpk, \mathbf{PK}, \mathbf{SK}\}$. Given a security parameter λ and generates a group public key gpk, a vector of public keys $\mathbf{PK} = (pk_1, \cdots, pk_n)$, and a vector of secret keys $\mathbf{SK} = (sk_1, \cdots sk_n)$;
- *Share Signing Algorithm:* $\mathsf{ShareSig}(sk_i, m) \rightarrow s_i$. Given a message m and a secret key share sk_i, the deterministic algorithm outputs a signature share s_i;
- *Share Verification Algorithm:* $\mathsf{ShareVerify}(m, (i, s_i)) \rightarrow 0/1$. Given a message m, a signature share s_i and an index i of the signer, this deterministic algorithm outputs 1 or 0 depending on whether s_i is a valid signature share generated by P_i or not;
- *Signature Combining Algorithm:* $\mathsf{Combine}(m, \{(i, s_i)\}_{i \in K}) \rightarrow \sigma/\perp$. Given a message m, and a list of pairs $\{(i, s_i)\}_{i \in K}$, where $K \subset [n]$ and $|K| = t$, this algorithm outputs either a signature σ for message m, or $\perp$ when $\{(i, s_i)\}_{i \in K}$ contains ill-formed signature share (i, s_i);
- *Signature Verification Algorithm:* $\mathsf{Verify}(m, \sigma) \rightarrow 0/1$. Given a message m and a signature σ, this algorithm outputs 1 or 0 depending on whether σ is a valid signature for m or not.

Hash functions. *Hash functions* are widely used in cryptography as one-way functions with a fixed output length. The hash function used in blockchains is usually a *cryptographic hash function* which satisfies both preimage and collision resistance.

B SECURITY ANALYSIS

Theorem 1 (Persistence). *If in a given* Kronos *round, an honest party P_i in shard S_c outputs a transaction tx at height k in shard ledger $S_c.\mathsf{log}_i$, then tx must occupy the same position in ledger $S_c.\mathsf{log}_j$ recorded by every honest party P_j in shard S_c.*

Proof. The persistence property relies on the majority honesty of shard configuration and safety of BFT deployed in each shard. During shard configuration/reconfiguration at the beginning of each epoch, each shard within n parties is configured to be majority-honest, with the proportion of Byzantine parties being less than $n/3$. Thus, the BFT protocols, which are tolerant against $n/3$ Byzantine parties in partial synchronous and asynchronous networks, and $n/2$ in a synchronous network, ensure safety and liveness successfully.

In a given shard S_c employing such a secure BFT protocol with an external function for transaction verification similarly, if an honest party outputs transaction tx in a committed transaction set TXs_ℓ in a certain round ℓ, then any party in S_c outputting TXs'_ℓ must hold that $\mathsf{TXs}_\ell = \mathsf{TXs}'_\ell$ with tx included. Each honest party in S_c appends the shard ledger $S_c.\mathsf{log}$ upon receiving committed transactions from BFT irrevocably. Therefore, tx must occupy the same position whenever any honest party records it in its ledger. □

Theorem 2 (Consistency). *There is no round r in which there are two honest party ledger states log_1 and log_2 with transactions tx_1, tx_2 respectively, such that $tx_1.\mathrm{I} \cap tx_2.\mathrm{I} \neq \emptyset$.*

Proof. We prove this theorem by contradiction. Suppose that there exist two conflicting transactions tx_1 and tx_2 where $tx_1.\mathrm{I} \cap tx_2.\mathrm{I} = \widetilde{I}$, within $\widetilde{\mathrm{utxo}}$. According to the transaction types of tx_1 and tx_2, we analyze consistency as follows:

Consistency among spend-transactions (intra-shard transactions). If tx_1 and tx_2 are both S-tx and recorded in the same shard log, they must be committed for intra-shard requests and output by some BFT rounds. Because honest parties check whether there are conflicting transactions in each BFT round output TXs before recording, tx_1 and tx_2 cannot be committed in the same round BFT. In each BFT round, TxVerify examines S-txs with the signature sig and the utxo existence in UTXO, so the conflict arises only if $\widetilde{\mathrm{utxo}}$ is valid in UTXO during both BFTs in which tx_1 and tx_2 are committed. While UTXO is updated upon completion of a BFT round, the safety of BFT ensures that each honest party holds the same UTXO updated timely. No matter which transaction spends $\widetilde{\mathrm{utxo}}$ first, it will be removed from UTXO and no one can spend it again. Conflicting tx_1 and tx_2 cannot be committed in different BFT rounds. Also, S-tx only spend utxo managed by the current shard and any utxo belongs to one shard only, then conflicting S-txs in different shards are impossible. Therefore, there is no conflict between S-txs.

Consistency between spend- and finish-transactions. Suppose that types of conflict transactions tx_1 and tx_2 are S-tx and F-tx, respectively. According to TxVerify, a S-tx is verified valid only if S-tx.I $\subseteq$ UTXO. However, F-tx.I is from buffer. If $tx_1.\mathrm{I} \cap tx_2.\mathrm{I} = \widetilde{I}$, there must be a round where $\widetilde{\mathrm{utxo}} \in$ UTXO and $\widetilde{\mathrm{utxo}} \in$ buffer at the same time. According to Kronos, any transaction output is either added to UTXO or stored in the output shard buffer through m_{buffer}. No transaction output belongs to UTXO and buffer simultaneously. Therefore, there is no conflict between S-tx and F-tx.

Consistency between spend- and back- transactions. The inputs of B-txs are outputs of cross-shard spend-transactions, which are not managed by the current shard. Because S-tx can only spend utxo managed by the current shard, there is no conflict between S-tx and B-tx.

Consistency among finish-transactions. If there are two conflict F-tx tx_1 and tx_2, they must be recorded either with different ids or in different shards because each id corresponds to only one F-tx in any shard. F-tx inputs are stored in buffer with their corresponding ids, thus one input can only be transferred by a F-tx with a matched id. The conflict can happen only if some stored input corresponds to two different ids, which means some utxo is spent on two different requests by conflict S-txs. While it is proven that there is no conflict between S-txs, so there is no conflict between F-txs.

Consistency between finish- and back-transactions. F-tx and B-tx inputs are both from buffer where the inputs are stored with corresponding id and been removed once the transaction is recorded. In case the ids of conflict F-tx tx_1 and B-tx tx_2 are different, there must be some input corresponding to two different ids. It happens only if some utxo is spent on two different requests by conflicting S-txs. So there is no conflict between F-tx and B-tx with different id. In case the ids of conflict F-tx tx_1 and B-tx tx_2 are the same, F-tx$[id]$ is signed only if at least $n-f$ parties have received all inputs of req$[id]$ and stored them in buffer, while B-tx$[id]$ is committed after receiving $m_{\mathrm{RJ}}[id]$ indicating some input of req$[id]$ is signed unavailable. According to Kronos, an honest shard managing some req$[id]$ input either spends it or signs unavailability and must multicasts messages for rejection to all involved shards in case its belonging input is unavailable. Hence, F-tx$[id]$ and B-tx$[id]$ for the

same req$[id]$ do not exist. Therefore, there is no conflict between F-tx and B-tx.
Consistency among back-transactions. Similar to F-tx, each id corresponds to only one B-tx. If B-tx$[id]$ and B-tx(id') are in conflict, there must be some stored inputs from two conflicting S-txs. While it is proven that there is no conflict between S-txs, so there is no conflict between B-txs. □

Theorem 3 (Atomicity). *A cross-shard transaction request* req$[\gamma]$ *is either executed by all involved shards if it is valid, or comprehensively rejected by each involved shard without any fund movement.*

Proof. The atomicity property is ensured by waiting for integral inputs before commitment and the rollback mechanism achieved with back-transactions.
Atomicity in valid request execution. If req$[\gamma]$ is a valid transaction request with all inputs available, it is delivered to all involved shards after submission to its output shard. Upon receiving req$[\gamma]$ and verifying input availability, each input shard constructs a spend-transaction S-tx$[\gamma]$ to spend the required inputs. The valid transaction S-tx$[\gamma]$, with available inputs, is committed by a certain round of BFT and executed by each input shard. The expenditure of each input shard is certified to the output shard with a certificate, and output shard stores verified inputs in its buffer. Upon storing integral inputs of req$[\gamma]$ in buffer, each honest party signs to execute req$[\gamma]$. The funds in buffer are accessed by $n - f$ valid signatures and transferred to the payee's address, finalizing the execution. Therefore, the valid transaction request is executed by all input and output shards.
Atomicity in invalid request rejection. In the case of an invalid request req$[\gamma]$, if it exhibits an incomplete structure, the output shard ignores it directly, and no further execution occurs by any shard, ensuring atomic rejection. Otherwise, the invalid req$[\gamma]$ is well-structured and delivered to all involved shards. The input shard managing an unavailable input verifies req$[\gamma]$ upon receiving it, preventing the execution of req$[\gamma]$ through any transaction but signing a threshold signature σ^{RJ} in a reject-message m_{RJ} to inform other involved shards of its invalidity. Upon receiving m_{RJ}, other input shards cease executing req$[\gamma]$ and remove the corresponding spend-transaction S-tx$[\gamma]$ from the waiting queue Q if it exists. In case the m_{RJ} is delayed and some input shard has "misexecuted" req$[\gamma]$, it corrects by constructing and committing a back-transaction B-tx$[\gamma]$ to pay back the spent input to initial address with the received signature σ^{RJ} in m_{RJ} as T-SIG. It is impossible for an honest shard to reject req$[\gamma]$ with a $(n-f, n)$-threshold signature σ^{RJ} in a reject-message m_{RJ} while spending to req$[\gamma]$ certified by cert within signatures of a majority shard members in a Buffer-Message m_{BF} simultaneously (because only at most f Byzantine parties might vote to spend and sign unavailability for The same request ambiguously). Therefore, no input of req$[\gamma]$ is transferred to the payee by the output shard, whose buffer never stores enough inputs due to a certificate loss of the unavailable input expenditure. Upon receiving verified reject-message m_{RJ} indicating req$[\gamma]$ invalidity, each honest party in the output shard empties req$[\gamma]$ inputs stored in buffer, and invalid req$[\gamma]$ is rejected by every involved shard completely. □

Theorem 4 (Liveness). *If a transaction request* req$[\gamma]$ *is submitted, it would undergo processing within κ rounds of communication (intra- or inter-shard), resulting in either a ledger-recorded transaction or a comprehensive rejection, where κ is the liveness parameter.*

Proof. The liveness property is guaranteed by the introduced submission paradigm and intra-shard BFT liveness.
Liveness in valid request processing. If req$[\gamma]$ is a valid request, the output shard forwards it to all involved shards. Every honest input shard verifies the request and creates a spend-transaction S-tx$[\gamma]$ to spend available inputs. As shards select transactions for BFT from the waiting queue Q in order, S-tx$[\gamma]$ is popped out and proposed within a limited time. Due to the liveness of BFT, S-tx$[\gamma]$ is eventually output after some rounds (referred to as κ_{BFT} rounds) of intra-shard communication. For an intra-shard request, the processing concludes, with each honest party in the shard recording it in the shard ledger, and the liveness parameter κ holds that $\kappa = \kappa_{\mathsf{BFT}}$. In the case of a cross-shard request, every input shard transmits the certificate of input expenditure on req$[\gamma]$ to the output shard's buffer in a Buffer-Message m_{BF} during a round inter-shard communication. The buffer eventually stores all inputs of the valid request, and every honest party in the output shard signs to the validity of req$[\gamma]$ during a round of intra-shard communication. The finish-transaction F-tx$[\gamma]$ is constructed after collecting $n - f$ valid signatures from distinct parties and then committed by BFT within κ_{BFT} rounds of intra-shard communication. Therefore, the liveness parameter holds that $\kappa = 2\kappa_{\mathsf{BFT}} + 2$.
Liveness in invalid request processing. If all involved shards of invalid req$[\gamma]$ are on good networks, only a round of intra-shard communication for signing the unavailable input and a round of inter-shard communication for invalidity declaration in reject-message m_{RJ} are required. The liveness parameter in this good-case scenario is $\kappa = 2$. In the worst case, if the reject-message m_{RJ} is delayed and some input has been expended through a BFT within κ_{BFT} rounds of intra-shard communication, the input shard gets back the input in a back-transaction B-tx$[\gamma]$ within another κ_{BFT} rounds of intra-shard communication. In total, the worst-case liveness parameter is $\kappa = 2\kappa_{\mathsf{BFT}} + 1$.
In summary, a submitted request must be processed within κ rounds of communication intra- or inter-shard, where $\kappa \leq 2\kappa_{\mathsf{BFT}} + 2$ with κ_{BFT} representing the limited round number of intra-shard communication during a BFT execution. □

C COMPLEXITY ANALYSIS

Lightweight cross-shard communication and message complexity (CS-COF & CS-MO). Coordination across shards ensures cross-shard transaction secure processing. During a certain *x-in-y-out* request processing, the specific output shard receives and transmits it to all involved shards (if it is well-structured), resulting in the number of inter-shard messages with $x + y - 1$. Each input shard spends available inputs for the request with a certificate and informs every output shard within an inter-shard Buffer-Message m_{BF}, adding xy to the CS-COF. The follow-up steps, executing the request with a transaction commitment and recording to the ledger, only require operations inside the output shards. Hence, the overall CS-COF for processing a valid transaction is $x + y - 1 + xy$, a value that is lower than $x(x + y - 1)$ in systems employing 2PC or relay mechanisms. This condition holds true when $(x-1)^2 > y$, a requirement often met because the number of output shards is typically 1 or 2.

For a well-structured but invalid request, it is also conveyed through $x+y-1$ inter-shard messages from the output shard. If any input shard verifies the request with an unavailable input, each honest party in the shard signs to create a threshold signature serving as proof of invalidity, transmitting the proof to all other involved shards. Consequently, $x+y-1$ reject-messages are dispatched. Assuming there are k input shards managing unavailable inputs, at most $k(x+y-1)$ reject-messages are transmitted in the worst case where all k shards experience a poor network or are subject to attacks. Regardless of whether some input has been spent on the invalid request or not, the subsequent steps for rejection are intra-shard operations. The value of CS-COF is such that CS-COF= $t(x+y-1)$ where the integer t lies in the range $2 \leq t \leq k+1$.

The messages transmitted across shards encompass both Buffer-Message, which pertain to available input spending, and Reject-Message, which convey statements about invalid requests. Alongside the processed request IDs, a Buffer-Message m_{BF} is comprised of a certificate $\mathsf{cert}(\mathsf{proof}, \mathsf{rt}, \mathsf{hp})$, where proof is acted by a threshold signature for the tree root rt, and hp signifies the hash path from a leaf node to the tree root. The length of the adopted hash function H output and the threshold signature are exclusively linked to the system security parameter λ, resulting in $|\mathsf{proof}| = |\mathsf{rt}| = O(\lambda)$. In a sharding blockchain system with a total of m shards, the maximum number of leaf nodes in a Merkle tree is $m-1$, making hp inclusive of at most $\log(m-1)$ hash values. Consequently, the overall message overhead of a Buffer-Message is $O(\lambda \log m)$. Nevertheless, as a single Buffer-Message provides certification for multiple requests processed together through a b-batchsize BFT with an identical output shard, the message overhead for a single request is $O(\frac{m\log m}{b}\lambda)$.

The batch size b is contingent on the performance of the BFT protocol and the bandwidth of the involved parties, typically ranging in the order of hundreds to tens of thousands in practical scenarios. The number of shards, however, is generally not excessively large, as an overly sharded system could compromise security by rendering each shard too small to withstand Byzantine adversaries. The inequality $\frac{m\log m}{b} < 1$ is easily satisfied, even in a large-scale system with hundreds of shards, with the batch size b typically reaching the thousand-level (where the block size for a BFT output is approximately 8KB when there are 100 shards, considering a common transaction length of $|tx| = 250$ bytes). Consequently, the message complexity $O(\frac{m\log m}{b}\lambda)$ is usually much lower than the values of $O(\lambda)$ or $O(\lambda \log b)$ found in state-of-the-art works. This configuration achieves lightweight inter-shard cooperation without compromising security or incurring additional intra-shard overhead.