Kronos: A Secure and Generic Sharding Blockchain Consensus with Optimized Overhead

Yizhong Liu, Andi Liu, Yuan Lu, Zhuocheng Pan, Yinuo Li, Jianwei Liu, Song Bian, Mauro Conti

***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. Currently, there is a lack of a generic sharding consensus pattern that achieves both security and low overhead.**

In this paper, we present Kronos, a secure sharding blockchain consensus achieving optimized overhead. In particular, we propose a new *secure sharding consensus pattern*, based on a *buffer* managed jointly by shard members. Valid transactions are transferred to the payee via the buffer, while invalid ones are rejected through happy or unhappy paths. Kronos is proved to achieve *security with atomicity* under malicious clients with *optimal intra-shard overhead* $k\mathcal{B}$ (k for involved shard number and $\mathcal{B}$ for a Byzantine fault tolerance (BFT) cost). Efficient rejection even requires no BFT execution in happy paths, and the cost in unhappy paths is still lower than a two-phase commit. Besides, we propose secure cross-shard certification methods based on *batch certification* and *reliable cross-shard transfer*. The former combines *hybrid trees* or *vector commitments*, while the latter integrates *erasure coding*. Handling b transactions, Kronos is proved to achieve *reliability* with low *cross-shard overhead* $\mathcal{O}(nb\lambda)$ (n for shard size and λ for the security parameter). Notably, Kronos imposes no restrictions on BFT and does not rely on time assumptions, offering optional constructions in various modules. Kronos could serve as a universal framework for enhancing the performance and scalability of existing BFT protocols, supporting generic models, including asynchronous networks, increasing the throughput by several orders of magnitude.

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 scales the consensus nodes to thousands, achieving a substantial throughput of 320 ktx/sec with 2.0 sec latency. Compared with the past solutions, Kronos outperforms, achieving up to a 12× improvement in throughput and a 50% reduction in latency when cross-shard transactions dominate the workload.

I. 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 exceptional properties of decentralization, immutability, and transparency, blockchain has become a promising instrument for cryptocurrencies, financial services, federated learning [41], privacy-preserving computation [4], and decentralized identity [42] to improve overall security and performance. Blockchain also drives the development of emerging domains such as Web3.0 [33] and Metaverse [28], requiring higher throughput and lower latency.

Scalability bottleneck and the potential of sharding. Practical applications reveal a significant challenge in traditional blockchains—*poor scalability* [40]. Each transaction requires submission to the whole network, and all participants verify each transaction through a consensus protocol, where *Byzantine fault tolerance* (BFT) is usually adopted. As participants scale, heavy communication and computation overhead reduces throughput and brings increased latency.

Elastico [40] pioneers sharding blockchain, leveraging sharding technology from the database field to address scalability issues. The sharding paradigm partitions all parties into a few smaller groups, referred to as *shards*, and each shard's major workload is to process a subset of transactions. Hence, each party only has to participate in some small shards, preserving lower *computation*, *communication*, and *storage overhead* despite the total node number in the system. Scaling a sharding blockchain to include more participants and shards holds the promise of achieving higher transaction throughput, as multiple shards increase processing parallelism [36]. Currently, significant attention has been directed towards blockchain sharding, with notable examples including [32], [52], [12], [20], [35], [3], [26], [55], [27], [48], [53], [49].

Sharding technology offers a promising solution for scalability enhancement, but it also comes with specific challenges. Each transaction is processed solely by one (or multiple) of all shards. As a result, ensuring a majority of honest nodes in each shard is crucial, requiring Byzantine node proportion to fall within the adopted BFT protocol's fault tolerance threshold. Consequently, methods for secure shard configuration have become a pivotal area of research, with notable works such as Omniledger [32] (SP'18), RapidChain [52] (CCS'18), and Gearbox [15] (CCS'22) offering remarkable solutions.

Introduced a new scenario: cross-shard transactions. In particular, 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) or accounts associated with the shard, bringing in *cross-shard transactions* [32] where input and output addresses belong to different shards. The 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*. Due to the state isolation across shards, cross-shard transactions cannot be processed by a single shard solely; they require multiple-shard cooperation, completing collaborative processing and consistent state updates.

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 [34].

Cross-shard transaction processing dominating security and efficiency. Cross-shard transaction processing involves state updates to multiple related shards, thus sharding blockchains need to satisfy *atomicity*, in addition to the *persistence*, *consistency*, and *liveness* applicable to regular blockchains. Atomicity requires that transaction execution follows an *"all-or-nothing"* principle, i.e., either all involved shards commit it, or each operation is aborted.

Notably, multiple inputs of cross-shard transactions may originate from different clients, a scenario common in *consolidated payments* or *crowdfunding transactions*. This can result in situations where certain inputs are invalid when malicious clients attend. As shown in Fig. 1, inputs of tx_α are both *available* (utxo_1, utxo_2 with verified signatures sig_1, sig_2), requiring *"all"*-execution, where utxo_1 and utxo_2 are both spent for tx_α. Failing to spend any input for a committed transaction compromises security, leading to an imbalance where the output value exceeds inputs and a double-spending risk. Conversely, if a request is incomplete or contains errors (e.g., missing/invalid signatures, or refers to a non-existent utxo' as tx_β in Fig. 1), it is considered invalid and proceeded with *"nothing"*. No values should be transferred (e.g., utxo_3 is not spent by tx_β). If atomicity is violated, two cases may occur. First, the output shard executes while the input shard does not, resulting in incorrect system balances. Second, only the input shard executes, leading to incorrect spending or permanent locking of honest client's money. Thus, ensuring secure and atomic cross-shard transaction processing is crucial, relying on rigorous consensus invocation in each shard and meticulous cross-shard cooperation with reliable message transfer.

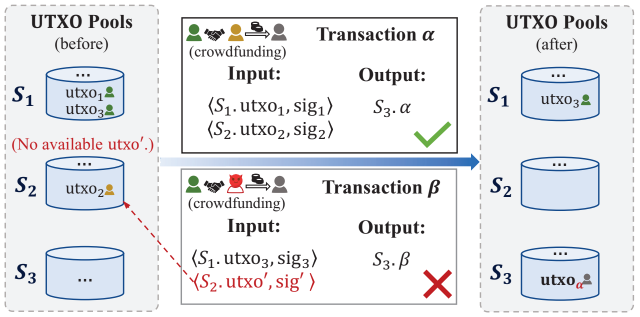


Fig. 1. Cross-shard transaction atomic execution.

Meanwhile, cross-shard transactions account for the vast majority of all transactions [52], so the *efficiency* of processing methods significantly influences the system performance. Primary costs include two aspects: *intra-shard overhead* and *cross-shard overhead*. The former primarily focuses on the expenses of the adopted BFT and the BFT invocation times required to process a cross-shard transaction (batch). Reducing BFT invocation times and implementing batch processing is crucial for mitigating intra-shard overhead. Furthermore, cross-shard overhead revolves around the transfer of transactions and proofs between shards. Applying batch proofs and reducing cross-shard message volumes could lower cross-shard overhead. Hence, a critical objective is to optimize both the intra-shard and cross-shard overheads for cross-shard transaction processing without compromising security and atomicity.

A. Remaining Issues of Prior Solutions

Q1: High overhead and compromised security without atomicity. The pioneering works of Omniledger [32], Chainspace (NDSS'18) [3], ByShard [24], and Haechi (NDSS'24) [53] commit cross-shard transactions through *two-phase commit* (2PC): *prepare* and *commit*. In a *prepare* phase, each input shard executes BFT to either lock available inputs or prove an unavailable input, with a certificate sent to other involved shards. In a *commit* phase, each input shard executes BFT again, spending or unlocking the locked inputs based on transaction validity. Compared with normal transaction processing with one BFT execution, 2PC's twofold BFT increases overhead (Table I, IS-Overhead). Subsequent studies, RapidChain [52], Monoxide (NSDI'19) [48], and Reticulum [49] (NDSS'24), decrease the high overhead of 2PC by eliminating the locking mechanism. Available inputs are spent directly to the payee. While this approach facilitates processing valid transactions at a lower cost, it destroys atomicity for invalid transaction execution (Table I, Atomicity).

Meanwhile, *malicious clients* pose a challenge in BFT [11], [30], [16], yet it has not received sufficient attention in sharding blockchains. A malicious client of a crowdfunding transaction can destroy security by either withholding the certificates in a client-driven 2PC [32] (i.e., *withholding attack*) or not sending requests to some input shards initially (i.e., *silence attack*, Section II, Challenge 1). Following works [3], [24] mitigate the risk by sending certificates among shards directly, but cannot prevent silence attacks. Some works try to address the issues by introducing additional dependency or special shard structures, leading to extra costs or new issues. *Reference shards* [12], [13] are adopted to transfer and commit cross-shard transactions through BFT consensus, resulting in high extra cost. *Cross-shard nodes* joining multiple shards is adopted [26], [25], while ensuring that each cross-shard transaction corresponds to a cross-shard node for processing is challenging. *Merging shards* after each epoch [29] is used to handle cross-shard transactions, yet it can lead to difficulties in achieving low-latency and responsive processing between two shards when shard number increases (Table I, Dependency).

Q2: Verbose cross-shard messaging and unreliable cross-shard transfer. In addition to the absence of secure and efficient processing patterns, cross-shard messaging also presents issues that need to be re-evaluated. Input shards need to send a large amount of valid cross-shard requests and certificates to relevant shards. Besides, cross-shard transactions are challenging to batch process effectively, leading to large volumes of cross-shard certificates. If each input is proved by a certificate [3], [5], [12], then b transactions require b proofs for transmission. Bundling cross-shard transactions with the same output shard into one block and signing the block as a certification [26], [48] compress the certificate size. However, this method ignores the transaction index size, and requires each shard to pack transactions as per output shards, disrupting normal processing order and bringing extra latency without responsiveness (Table I, CS-Overhead).

Furthermore, existing solutions fail to ensure the reliability of critical cross-shard message transfer (Table I, Reliability). In a Byzantine environment, it is essential that certificates are ensured to be sent by a source shard and reliably received by at least one honest party in a destination shard. However,

TABLE I. COMPARISON OF KRONOS WITH STATE-OF-THE-ART SHARDING BLOCKCHAIN PROTOCOLS

System	IS-Overhead[†]	CS-Communication (for b transactions)[‡]			Dependency	Atomicity[◇]	Genericity[⋆]
		Paradigm	CS-Overhead	Reliability[ℶ]			
Omniledger [32]	$2k\mathcal{B}$	leader-to-client-to-leader	$\mathcal{O}(b(\log b + \lambda))$	✘(HL, HC)	HL, HC	✔✘	p. sync.
ByShard [24]	$2k\mathcal{B}$	$\mathcal{O}(1)$-to-$\mathcal{O}(1)$, $\mathcal{O}(n)$ rounds	$\mathcal{O}(n^2 b\lambda)$	✘(timers)	timers, HC	✔✘	sync.
RapidChain [52]	$k\mathcal{B}$	leader-to-all-to-all	$\mathcal{O}(n^2 b\lambda)$	✘(HL)	HL	✔✘	sync.
Sharper [5]	-[♭]	leader-to-all-to-all	$\mathcal{O}(n^2 b\lambda)$	✘(HL)	HL	✔✘	p. sync.
Monoxide [48]	$k\mathcal{B}$ $(k = 2)$[§]	miner-to-$\mathcal{O}(n)$[§]	$\mathcal{O}(nb\lambda)$	✘(HM)	HM, fee incentive	✘	p. sync.
Chainspace [3]	$2k\mathcal{B}$	$\mathcal{O}(n)$-to-$\mathcal{O}(n)$	$\mathcal{O}(n^2 b\lambda)$	✔	HC	✔✘	p. sync.
AHL [12]	$(2k + 3)\mathcal{B}$	$\mathcal{O}(n)$-to-$\mathcal{O}(n)$	$\mathcal{O}(n^2 b\lambda)$	✔	TEE, HC	✔✘	p. sync.
Pyramid [26]	$(k + 1)\mathcal{B}$[ℵ]	$\mathcal{O}(n)$-to-$\mathcal{O}(n)$[ℵ]	$\mathcal{O}(n^2 b\lambda)$	✔	shard overlap	✔✘	p. sync.
Kronos (our work)	<u>$\boldsymbol{k\mathcal{B}}$</u>	$\boldsymbol{\mathcal{O}(n)}$**-to-**$\boldsymbol{\mathcal{O}(n)}$**+EC code**	$\boldsymbol{\mathcal{O}(n\xi\lambda)}$[♮](HT-RCBC) $\boldsymbol{\mathcal{O}(nb\lambda)}$ (VC-RCBC)	✔	–	✔	sync. / p. sync. /**async.**

[†] "IS" and "CS" denote *intra-shard* and *cross-shard*. n denotes shard size (all shards are the same size). k denotes the number of shards (input and output shards have no intersection) involved in a cross-shard transaction. $\mathcal{B}$ denotes a BFT cost, which could be $\mathcal{O}(n)$ in p.sync. BFT Hotstuff [50] and $\mathcal{O}(n^2)$ in async. BFT such as FIN [18].

[‡] λ and m respectively represent the security parameter and the total number of shards within the system.

[ℶ] "HL", "HC" and "HM" are abbreviation of "honest leaders", "honest clients", and "honest miners", respectively.

[◇] ✔✘ indicates that the system operates with *weak atomicity*, relying on certain assumptions.

[⋆] "sync.", "p. sync." and "async." are abbreviation of *synchronous, partially synchronous* and *asynchronous*.

[ℵ] Pyramid claims to reach overall consensus among shards through rounds of voting, with IS-overhead equal to $k + 1$ times BFT. It also discusses a leader-to-leader paradigm with a CS-Overhead of $\mathcal{O}(kb\lambda)$ where reliability relies on HL.

[♭] In Sharper, all communication for cross-shard transaction processing is conducted across shards.

[§] Monoxide operates as a miner-driven *Proof-of-Work* (PoW) based system, exclusively considering single-input, single-output cross-shard transactions.

[♮] HT-RCBC (VC-RCBC) denotes hybrid tree (vector commitment) reliable cross-shard batch certification. $\xi = \max(n\log m, n\log n, b)$. In sharding settings (e.g., $m = 50$, $n = 64$ and $b = 10\text{k}$), $\mathcal{O}(n\log m), \mathcal{O}(n\log n) < \mathcal{O}(b)$ is commonly satisfied. $\mathcal{O}(n\xi\lambda)$ approximate $\mathcal{O}(nb\lambda)$ when b is larger than $n\log m$ and $n\log n$.

existing solutions either delegate this task to shard leaders [32], [26], [5], which can fail facing a *disguise attack* (Section II, Challenge 2), or resort to a rude "all-to-all" broadcast without further optimizations [3], [12], [52], resulting in high communication overhead $\mathcal{O}(n^2 b\lambda)$ (Table I, CS-Overhead).

Q3: Lack of research on generic model sharding blockchains. A *generic model* sharding blockchain requires elaborate and non-trivial designs for secure and reliable *asynchronous* handling of intra-shard transactions, cross-shard proof construction, and cross-shard message transfer. It cannot be achieved simply by combining an asynchronous BFT with conventional cross-shard processing methods.

The above issues Q1-Q3 expose an open question lying in the design space of sharding blockchains:

Can we design a generic sharding blockchain consensus achieving security and efficiency with optimized overhead?

B. Our Contributions

We affirmatively address the aforementioned question by introducing Kronos, a generic sharding consensus that ensures security with atomicity and realizes optimized overhead.

A new sharding consensus pattern realizing security with optimal intra-shard overhead. We propose a *new paradigm* for cross-shard transaction processing based on (1) *request delivery* and (2) *buffer* mechanisms. Request delivery enables the output shard to perform request validation and reliable forwarding, preventing silence attacks. The buffer mechanism, based on a *group address* jointly maintained by all parties of an output shard, securely manages spent inputs from other shards in cross-shard transactions. For valid transactions, the input amounts are transferred to the payee through the buffer. For invalid transactions, in happy paths, comprehensive rejection is achieved through cross-shard multicasting. In unhappy, the overhead remains superior to 2PC. Notably, Kronos is proved to achieve *optimal intra-shard overhead* $k\mathcal{B}$ (Table I, IS-Overhead), our newly proposed metric for measuring intra-shard complexity leveraging BFT invocation times. Efficient rejection even requires no BFT execution in happy paths. The cost in unhappy paths is still lower than 2PC. Kronos is proven to satisfy security with *atomicity* under malicious clients and leaders (Table I, Atomicity).

Secure cross-shard certification methods realizing reliability with low cross-shard overhead. We propose reliable cross-shard certification utilizing (1) *batch certification* and (2) *reliable cross-shard transfer* mechanisms. Batch certification proves multiple inputs in a single proof through *hybrid trees* or *vector commitments* without compromising *responsiveness*. Shards process requests and construct a hybrid tree after BFT, where each leaf node is computed from one output shard's requests. Optionally, as the node number increases, vector commitment utilizes one output shard's requests as a vector point, reducing the cross-shard proof number for b transactions from $\mathcal{O}(b)$ without batch to $\mathcal{O}(1)$. Further, reliable cross-shard transfer adopts $\mathcal{O}(n)$-to-$\mathcal{O}(n)$ multicasting with *erasure coding*, encoding certified requests into the hybrid tree or vector. Each node only needs to send one code block and proof. In particular, Kronos is proved to realize cross-shard message transmission *reliability* (Table I, Reliability) with low cross-shard overhead $\mathcal{O}(nb\lambda)$ (Table I, CS-Overhead).

Generic sharding blockchain constructions supporting asynchronous networks. Kronos imposes no restrictions on the BFT employed in each shard and does not rely on any time assumption. We provide an independent transaction verification interface invoked by intra-shard (partially) synchronous or asynchronous BFT. Besides, Kronos offers *batch-proof-after-BFT* for asynchronous networks, and *batch-proof-with-BFT* for (partially) synchronous networks. Further, Kronos provides $\mathcal{O}(n)$-to-$\mathcal{O}(n)$ for asynchronous networks and $\mathcal{O}(1)$-to-$\mathcal{O}(1)$ with *cross-shard view-change* for (partially) synchronous network to realize reliable cross-shard transfer. Notably, Kronos could serve as a universal framework for enhancing the performance and scalability of existing BFT protocols, supporting generic models (Table I, Genericity), making it the first asynchronous sharding blockchain. Kronos scales the consensus nodes to thousands and increases the throughput by several orders of magnitude.

A large-scale implementation of specific constructions. To demonstrate the practical performance of Kronos, we implement it using an asynchronous BFT, Speeding Dumbo [22], 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 320 ktx/sec with a network size of 1000 nodes, and the latency is 2.0 sec. We compare Kronos with 2PC adopting the same BFT protocol. Kronos achieves up to 12× throughput improvement, with a halved latency. To demonstrate the generality, we also deploy Kronos with a partial synchronous BFT HotStuff [50] and evaluate the performance.

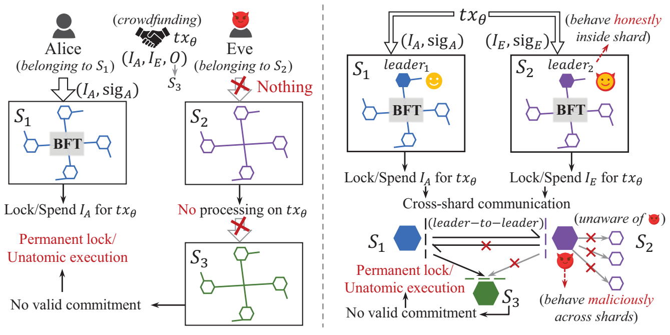
II. Challenges and Our Solution

Next, we dive deep into the specific challenges and give our high-level ideas in solving these issues.

Challenge 1: On processing pattern—Both *secure* and *low overhead* in *generic models* against *silence attack*. We identify *silence attack by malicious clients* (Fig. 2a). In crowdfunding transactions, inputs belong to multiple clients, including malicious ones (Eve). If submission is done by a single client, a malicious one can selectively send requests to a set of shards (S_1) while neglecting others (S_2). This results in S_2 being unaware of the transaction, leading to valid inputs (I_A) being incorrectly spent or permanently locked. The same issue arises when submission is done by respective input-holding clients to their corresponding shards.

In generic models, designing secure and low-overhead processing patterns resilient to silence attacks is non-trivial. Firstly, if multiple clients obtain each other's signatures beforehand and then submit to all relevant shards individually, it increases the client load. Employing *timeouts* by relevant shards to unlock is suitable for (partially) synchronous networks, yet is inapplicable to asynchronous ones. Shards cannot distinguish whether cross-shard proofs are delayed or transactions are not processed by other shards, leading to inconsistency among shards. Secondly, input shards need to execute BFT two times in 2PC. Removing one to reduce overhead is impractical since both phases involve status changes: from "available" to "locked", and then from "locked" to "unlocked" or "spent". Directly removing the locking mechanism violates atomicity, where valid inputs are directly spent, leading to partial execution of transactions when there are invalid inputs. The inability to achieve state rollback compromises security.

Challenge 2: On cross-shard certification—Both *reliable* and *efficient* in *asynchronous networks* against *disguise attack*. We identify a *disguise attack* by malicious leaders (Fig. 2b). In most schemes, the leader acts as the coordinator for cross-shard message transmission. A malicious leader ($leader_2$) might behave honestly within the shard but acts maliciously across shards: it does not send crucial messages



(a) Malicious client silence attack (b) Malicious leader disguise attack

Fig. 2. Attacks from malicious clients and shard leaders.

(proof) to relevant shards and does not forward messages from other shards to the nodes within its shard (S_2). Consequently, nodes within the shard are unaware of messages from other shards and remain oblivious to the leader's misconduct, resulting in the permanent locking of relevant inputs (I_A) and undermining atomicity and liveness.

In asynchronous networks, designing reliable and efficient cross-shard certification against disguise attacks is challenging. Firstly, setting timeouts by shard nodes, and initiating view-changes if relevant cross-shard messages are not responded within the timeout period is suitable for (partially) synchronous networks but not for asynchronous ones. Secondly, employing n-to-n cross-shard broadcasting incurs significant cross-shard communication overhead $O(n^2 b\lambda)$ (b for transaction number), which limits the scalability. Thirdly, cross-shard transactions are challenging to batch process and batch certify, leading to a high volume of proofs. Trivially packing blocks according to output shards destroys responsiveness and disrupts the transaction processing order. Besides, in asynchronous networks, consensus is non-deterministic, making it difficult to directly use consensus quorum certificates as cross-shard proofs.

Our solutions from a high-level perspective. Now, we walk through how we address the challenges.

Reliable paradigm for request submission. To prevent silence attacks from malicious clients, we implement a gathering-style submission paradigm for cross-shard transaction requests. Each client involved in a request is required to submit their signatures to the output shard, which checks the request integrity and transfers it to the involved shards. This ensures all relevant shards receive and process the transaction, further sending cross-shard proofs to achieve atomic processing.

A brand new secure sharding consensus pattern with optimal intra-shard overhead. We design a novel pattern for cross-shard transaction processing that achieves security and optimality. Initially, input shards invoke BFT to deposit available inputs into the "buffer" of the output shard, collectively managed by of the output shard nodes. For valid transactions, all transaction inputs are available; the output shard executes BFT to transfer the amount from the buffer to the payee. Consequently, processing valid transactions only requires invoking BFT k times (where k denotes the number of shards involved in a transaction). For invalid transactions, the input shard adopts cross-shard multicasting instead of BFT to transmit unavailability proofs. At this point, if the other available inputs have not been spent by other input shards (*happy path*), both the input and output shards discard the request. Otherwise, if the other inputs have already been spent (*unhappy path*), the output shard removes the input from the buffer, while the input shard executes BFT to roll back the spent input.

How to realize the buffer? Making an individual entity the managers of the buffer is evidently insecure. Therefore, we enable the entire output shard to collectively manage the buffer through a $(f+1,n)$ threshold signature (e.g., BLS threshold signature [9]). The honesty of each output shard ensures the security of the buffer. Note that for the sake of generality, asynchronous BFT is supported in intra-shard consensus, which typically also requires BLS threshold signatures to generate randomness. Hence, the initialization cost of threshold signatures has not been additionally incurred. Alternatively, multi-signatures [8] could be adopted to reduce the initialization overhead.

Batch certification for cross-shard transactions. To address the challenge of heavy cross-shard communication, we introduce a novel batch mechanism that proves multiple input availability through one certificate. Each shard operates normally, running BFT to process transactions in the order of their arrival. After transactions are committed, honest parties classify them based on their output shards and generate a certificate for each transaction set with the same output shard through a Merkle tree or a vector commitment. This "*batch-proof-after-BFT*" approach reduces the proof number for b transactions from $\mathcal{O}(b)$ to $\mathcal{O}(1)$ while ensuring responsiveness, and is applicable to asynchronous networks.

Reliable cross-shard transfer with low cross-shard overhead. In addition to sending proofs, the IDs of the certified b transactions in the batch need to be sent. To achieve asynchronous reliability, $\mathcal{O}(n)$*-to-*$\mathcal{O}(n)$ cross-shard message transfer is necessary. Thus, batch certification can reduce the complexity from $\mathcal{O}(n^2 2b\lambda)$ to $\mathcal{O}(n^2(b+1)\lambda)$. Kronos employs *erasure-coding* to decrease the overhead. The request statements are encoded to n code blocks, and each party is only responsible for sending one code block ($\mathcal{O}(b/n\lambda)$) to parties in the output shard. To prevent Byzantine parties from sending fake code blocks, Kronos commits and proves the blocks as a lower *code tree* or *code vector*. Compared to a straightforward broadcast, the communication overhead of cross-shard batch certification decreases from $\mathcal{O}(2n^2b\lambda)$ to $\mathcal{O}(nb\lambda)$.

III. Problem Formulation

In this section, we describe cryptographic primitives, notations, system models, security goals, and performance metrics.

A. Cryptographic Primitives

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. After initial key generation by function $\mathsf{SigSetup}$, each node has a private function $\mathsf{ShareSig}$ and public functions $\mathsf{ShareVerify}$, $\mathsf{Combine}$ and Verify. Both the signature share and the combined threshold signature is *unforgeable* and the scheme is *robust* (a valid threshold signature on a message m is sure to be generated) [23]. See Appendix A for definitions.

Erasure coding. Let $0 \leq r \leq n$, a (r,n)-erasure coding scheme encodes a message M into n blocks using algorithm $\mathsf{ECEnc}(\varphi, M, n, r) \rightarrow \{a_i\}_n$. The original message M can be reconstructed through decoding algorithm $\mathsf{ECDec}(\varphi, T) \rightarrow M$, where T consists of at least r encoded blocks a_i. φ denotes the polynomial utilized for coding. The encoded results $\{a_i\}_n$ by any party remain consistent as long as the same φ is employed in $\mathsf{ECEnc}(\varphi, M, n, r)$, and M is certainly retrieved from r correct blocks and φ [39].

Merkle tree. A *Merkle 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. The construction is denoted as $\mathsf{TreeCon}(\text{data set}) \rightarrow (\mathsf{tree}, \mathsf{rt}, \mathsf{hp})$ in this paper, where hp denotes the hash path from a leaf node to the root.

Vector commitment. A *vector commitment* allows a prover to commit a vector $\boldsymbol{vec}$ of ℓ messages through algorithm $\mathsf{VecCom}(\boldsymbol{vec}) \rightarrow (C, aux)$ (where aux is auxiliary information used for proof generation), and each vec_i at position i has a proof $\Lambda_i \leftarrow \mathsf{ComOpn}(vec_i, aux)$. Any validator can verify the commitment at any position $i \in [\ell]$ of the vector through $\mathsf{ComVrf}(C, vec_i, \Lambda_i)$, i.e., reveal vec_i equals to the i-th committed message. The size of the commitments and proofs are concise and independent of ℓ [46].

B. Notations

Byzantine fault tolerance protocol. In most sharding blockchain systems, each shard achieves intra-shard consensus through *Byzantine fault tolerance* protocols (denoted as BFT). BFT ensures *safety* and *liveness* [10] [1] 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. Primarily, Kronos is applicable to both account and UTXO models. For ease of description, we adopt the UTXO model in this paper. The methods of transaction processing and validation can be adapted to the account model with minor adjustments. When processing client requests, shards construct transactions as tx where $tx = (\mathsf{type}, id, \mathbf{I}, \mathbf{O})$. type represents the type of the transaction. id denotes the transaction request ID. $\mathbf{I} = \{I_1, I_2, \cdots\}$ indicates the transaction input set, where each $I_i \in \mathbf{I}$ consists of the belonging shard S_i, unspent transaction output utxo_i, and the client's signature sig_i. $\mathbf{O} = \{O_1, O_2, \cdots\}$ denotes transaction output, where each $O_j \in \mathbf{O}$ includes the output shard S_j, payee's public key pk_j, and the output value v_j.

There are three types of transactions in Kronos: Spend-Transaction (denoted as SP-tx, within $\mathsf{type} = \mathsf{SP}$) for input shard spending, Finish-Transaction (denoted as FH-tx, within $\mathsf{type} = \mathsf{FH}$) for output shard committing, and Back-Transaction (denoted as BK-tx, within $\mathsf{type} = \mathsf{BK}$) for rolling back invalid execution.

[1]Also called *validity, agreement, and termination* in related work.

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 completed transactions to shard ledger log with the format that $\mathsf{log} = tx_1 \parallel tx_2 \parallel \cdots$. Cross-shard transactions are recorded in output shards' ledgers.

C. System model

Network model. Each party connects to each other through a peer-to-peer (P2P) network. The network is generic, including *synchronous*, *partially synchronous*, and *asynchronous* (which is with the weakest time assumption of all network models). In an asynchronous network, an adversary can casually delay messages or disrupt their order, but each message will be received eventually. Note that our transaction processing methods are designed to be applicable to all networks. In synchronous or partially synchronous networks, alternative methods could be selectively employed to enhance processing efficiency.

Adversary model. The adversary has the capability to fully control multiple Byzantine nodes, which can deviate from protocol specifications but are unable to forge signatures of honest nodes. Within each shard, the number of malicious nodes does not exceed the security threshold, a constraint ensured by the secure *shard configuration*.

Besides, the adversary controls *malicious clients* 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 II).

Shard configuration. Kronos operates within a secure shard configuration. Each shard is considered an *honest shard*, where the number of Byzantine parties f and shard size n satisfy $f/n < 1/3$ ($1/2$ in synchronous networks). A secure shard configuration could be realized by [52], [3], [12], [15], [54].

IV. Kronos

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

A. Problem Formulation

1) Security Goal: We define a secure sharding blockchain 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 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 in the same shard, 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: For a transaction request involving value transfer across multiple shards, all involved shards consistently either execute the required value-transferring operations in their entirety during commitment (valid requests), or comprehensively reject it without any final commitment or value transfer (invalid requests).*
- *Liveness: Once a transaction request is submitted to the system, it will be processed eventually, either executed through a committed transaction tx recorded in a shard ledger or rejected with corresponding proofs.*

2) Performance Metrics: Aiming at processing transactions in sharding blockchains at low costs, we propose critical efficiency metrics.

Intra-shard communication overhead (IS-ω, Definition 2): As most sharding blockchains reach consensus on transactions through respective BFT with various communication complexities in different systems, we denote $\mathcal{B}$ to abstract a BFT cost. For example, an intra-shard transaction is committed through one intra-shard BFT, where IS-$\omega = \mathcal{B}$. For a cross-shard one, if there are two input shards, each running BFT twice in two phases during 2PC, and one output shard running BFT once, then the total IS-ω is $5\mathcal{B}$.

Definition 2 (Intra-shard communication overhead)**.** *Intra-shard communication overhead IS-ω refers to the overhead within all involved shards during a single transaction processing cycle. For generality, $\mathcal{B}$ abstracts a BFT cost, and IS-ω uses $\mathcal{B}$ to measure the total number of BFT executions.*

Cross-shard communication overhead (CS-ω, Definition 3): CS-ω is determined by the adopted cross-shard (also referred to as inter-shard) communication paradigm and the size of the message being transmitted. The network environment is also influential, as achieving reliability is certainly more challenging in a poor asynchronous network compared with an ideal synchronous one, necessitating more messages as insurance.

Definition 3 (Cross-shard communication overhead)**.** *cross-shard communication overhead CS-ω refers to the total bits of messages transmitted across shards for cross-shard transaction processing.*

B. System Overview

System initialization. The whole system has N nodes, which are divided into m shards. Each shard S_i (where $i = 1, 2, \cdots, m$) involves a set of parties (nodes) $\{P_j\}_{j \in [n]}$, where n is the shard size and $[n]$ denotes the integers $\{1, 2, \cdots, n\}$. Each shard initializes two threshold signature schemes (or multi-signature) among the shard participants, where the threshold values are $\mathsf{T} = n - f$ (i.e., for BFT and invalidity proof generation) and $\mathsf{t} = f + 1$ (i.e., for buffer management), respectively. Each party P_j can get its individual secret keys, sk_j^{T} and sk_j^{t}, and corresponding public keys. The setup can be executed through *distributed key generation* (DKG) [1], [14], [7] (unnecessary when using multi-signature). Notably, the group public key of $(f+1, n)$-threshold signature scheme gpk^{t}

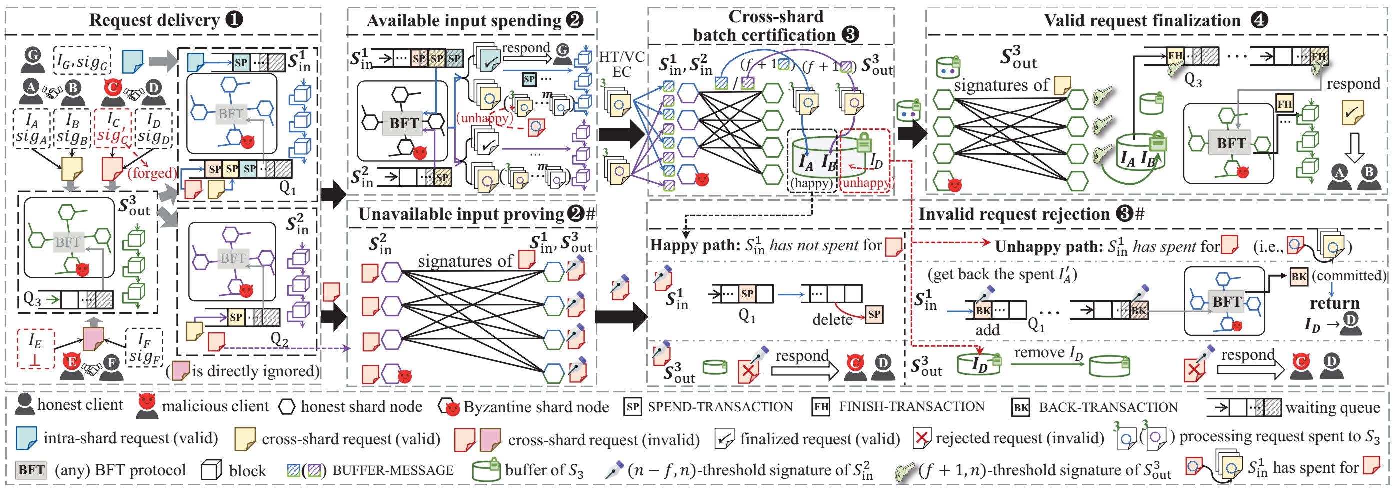

Request delivery ❶
Available input spending ❷
Cross-shard batch certification ❸
Valid request finalization ❹
Unavailable input proving ❷#
Invalid request rejection ❸#
Happy path: S^1_{in} has not spent for
Unhappy path: S^1_{in} has spent for
HT/VC EC
BFT
respond
(forged)
(unhappy)
(happy)
is directly ignored
signatures of
get back the spent I'_A
add
delete
remove I_D
(committed)
return I_D
honest client
malicious client
honest shard node
Byzantine shard node
SPEND-TRANSACTION
FINISH-TRANSACTION
BACK-TRANSACTION
waiting queue
intra-shard request (valid)
cross-shard request (valid)
cross-shard request (invalid)
finalized request (valid)
rejected request (invalid)
processing request spent to S_3
(any) BFT protocol
block
BUFFER-MESSAGE
buffer of S_3
$(n-f,n)$-threshold signature of S^2_{in}
$(f+1,n)$-threshold signature of S^3_{out}
S^1_{in} has spent for

Fig. 3. Transaction processing pattern of Kronos. Step ❶ - ❹ for valid cross-shard transactions. Step ❶, ❷ #, ❸ # for invalid ones where Step ❸ # includes two cases: happy and unhappy paths.

serves as the shard buffer address receiving cross-shard inputs. Each shard's gpk^{T} and gpk^{t} is public to all participants across the network. Besides, parties in the system share a set of m polynomials φ_c for the erasure coding of each shard S_c. 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 $\mathsf{TxVerify}(tx) = 1$.

Overview. Fig. 3 gives the transaction processing pattern of Kronos. Clients (honest A, B, D, F, G and malicious C, E) initially submit their requests to corresponding output shards (G submits an intra-shard request to S^1_{in}; A&B, C&D, E&F submit cross-shard requests to S^3_{out}). Intra-shard requests (G's) are processed by constructing and adding intra-shard SPEND-TRANSACTIONS (SP-tx for I_G of G's) to the BFT waiting queue. Incomplete cross-shard requests (E&F's) lacking necessary information are ignored by the output shard directly. Other cross-shard requests (A&B's and C&D's) are delivered to other involved shards (S^1_{in} and S^2_{in}) (Step ❶). Upon receiving a cross-shard request, each input shard processes it in an intra-shard way first. If inputs managed by the current shard are available (I_A, I_B of A&B's and I_D of C&D's), the shard constructs cross-shard SPEND-TRANSACTION (SP-tx for I_A and I_D in S^1_{in}, SP-tx for I_B in S^2_{in}) where the output address is the output shard (S^3_{out}) buffer, and adds it to the BFT waiting queue. BFT takes out transactions from the top of the queue and commits them in batches regardless of specific transaction types. Committed intra-shard SPEND-TRANSACTIONS (SP-tx for I_G) are recorded to the shard ledger and responded to corresponding clients by honest shard nodes (Step ❷). Cross-shard requests with committed cross-shard SPEND-TRANSACTIONS spent to buffers are certified to the corresponding output shards (S^3_{out}) in batch. Honest shard nodes collectively prove these request processing through hybrid tree or vector commitment technology optionally, and utilize erasure coding to amortize each node's message size when sending the proof within BUFFER-MESSAGES in the reliable $\mathcal{O}(n)$-to-$\mathcal{O}(n)$ cross-shard communication paradigm. After receiving a quorum of messages, honest nodes in the output shard verify the proof and store certified inputs (I_A, I_B, and in the unhappy scenario, I_D) in the shard buffer (Step ❸). Once the buffer has stored all inputs of a certain request (I_A, I_B of A&B's), each honest node signs to the request validity and broadcasts the signature inside the shard. When there are $f+1$ valid signatures, the honest nodes construct a FINISH-TRANSACTION (FH-tx of A&B's) to transfer the inputs from buffer to the real output address. Committed FINISH-TRANSACTIONS are also recorded to the shard ledger and responded to the clients (A and B) (Step ❹).

In case honest nodes in an input shard (S^2_{in}) find an unavailable input (I_C of C&D's), they do not construct any transaction but sign to the invalid request using their private keys of the $(n-f, n)$-threshold signature. The signatures are sent to all other involved shards (S^1_{in} and S^3_{out}) (Step ❷ #). Upon receiving a quorum of $n-f$ signatures, shards abort the request processing. Output shard (S^3_{out}) informs the request invalidity with the threshold signature to its clients (C and D). In the happy path, these signatures arrive at other input shards (S^1_{in}) *before* they spend for the request, where these input shards can simply abort the processing by removing corresponding SPEND-TRANSACTIONS (SP-tx for I_D) from their queues (happy path of Step ❸ #). In the unhappy path where signatures are received *after* some input shard spending for it, these shards construct and commit BACK-TRANSACTIONS (BK-tx for I_D) with the threshold signature to roll back the spent inputs (unhappy path of Step ❸ #).

C. Valid Transaction Processing of Kronos

Step ❶ : Request delivery (by clients and output shard).

Algorithm 1 shows the procedure of request submission in Step 1. Clients initiate the process by submitting transaction request $\mathsf{req}[id]$ to the output shard S_{out}. Once $\mathsf{req}[id]$ is complete with all necessary information, including signatures for each input, the output shard ID, payee's public key, and ensuring that the output value is less than the total inputs, it undergoes further process (Lines 1-5). If $\mathsf{req}[id]$ is cross-shard, members in S_{out} deliver it to all (at least $f+1$) members of 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 (Lines 6-7). In the case of an intra-shard request which involves transfer within S_{out} solely, $\mathsf{TxCon}_{\mathsf{SP}}$ is invoked (Line 8).

Step ❷ : Available input spending (from input shard to output shard buffer).

Algorithm 1 Request delivery (RD)

```
    Let clients submit every transaction request to parties of its output shard.
    ▶ As a party $P_i$ in shard $S_c$ (Step ❶ ):
 1: upon receiving a request $\mathsf{req}[id]$ submitted by clients do
 2:   verify that:
 3:   • $S_c$ serves as the output shard of $\mathsf{req}[id]$
 4:   • each input has a signature $\mathsf{sig}$
 5:   • the output value does not exceed the total value of inputs
 6:   if $\mathsf{req}[id]$ is a verified cross-shard request then
 7:     send $\mathsf{req}[id]$ to parties of other involved shards
 8:   else if $\mathsf{req}[id]$ is a verified intra-shard request, execute SPEND-TRANSACTION construction (TxCon$_{\text{SP}}$, Algorithm 2)
```

As shown in Algorithm 2, after receiving $\mathsf{req}[id]$, honest parties in the input shard S_{in} verify whether the required inputs managed by S_{in} are available. The verification involves checking $\mathsf{UTXO}_{\mathsf{in}}$ and validating the client signatures (Lines 1-4). Failed intra-shard request is refused to clients (Lines 5-6). For an invalid cross-shard request, a rejection message is transmitted to corresponding shards by invoking Algorithm 7 (Lines 7-8). If the conditions are satisfied, they proceed to construct a SPEND-TRANSACTION, SP-tx$[id]$, for the input expenditure (Lines 9-10). If $\mathsf{req}[id]$ is intra-shard, the output address pk is the payee's public key (Lines 11-12). In case $\mathsf{req}[id]$ is cross-shard, transferring funds to other shards, the output field $\mathbf{O}$ contains the output shard ID, the output shard buffer address $gpk^{\mathsf{t}}_{\mathsf{out}}$, and the transferred value (Line 13). Subsequently, SP-tx$[id]$ is added to the queue Q waiting for BFT commitment by Algorithm 3 (Line 14).

Algorithm 2 SPEND-TRANSACTION construction (TxCon$_{\text{SP}}$)

```
    ▶ As a party $P_i$ in shard $S_c$ (Step ❷ ):
 1: upon receiving $\mathsf{req}[id]$ do
 2:   verify that:
 3:   • input $I$ of $\mathsf{req}[id]$ managed by $S_c$ holds $I.\mathsf{utxo} \in \mathsf{UTXO}_c$
 4:   • $I.\mathsf{sig}$ is valid                    ▷ verify the input availability.
 5:   if some $I'$ verification fails and $\mathsf{req}[id]$ is intra-shard then
 6:     respond to the client that $\mathsf{req}[id]$ commitment failed
 7:   if some $I'$ verification fails and $\mathsf{req}[id]$ is cross-shard then
 8:     execute request rejection message transmission (RRMT, Algorithm 7)
 9: upon verified $\mathsf{req}[id]$ do
10:   construct SP-tx$[id] := (\text{SP}, id, \mathbf{I}, \mathbf{O})$, where $\mathbf{I}$ is set of $I$s to be spent
11:   if $\mathsf{req}[id]$ is intra-shard then
12:     set $\mathbf{O} := (S_c, pk, v)$ where $pk$ is the public key of payee  ▷ spend to the payee directly.
13:   otherwise, $\mathbf{O} := (S_{\mathsf{out}}, gpk^{\mathsf{t}}_{\mathsf{out}}, \mathbf{I}.v)$ where $gpk^{\mathsf{t}}_{\mathsf{out}}$ is the public key of the output shard buffer
14:   append SP-tx$[id]$ to Q and wait for BFT to commit it (Algorithm 3)
    ▶ SP-tx verification (invoked by Algorithm 3):
15: function $\mathsf{TxVerify}(tx)$
16:   if $tx.\mathsf{type} = \text{SP}$ then
17:     if for each $I_i \in \mathbf{I}$ where $I_i = \langle S_c, \mathsf{utxo}_i, \mathsf{sig}_i \rangle$, $\mathsf{utxo}_i \in \mathsf{UTXO}$ and $\mathsf{sig}_i$ is verified with $\mathsf{utxo}_i.pk$, return 1
18:     otherwise, return 0 and execute RRMT (Algorithm 7)
```

Algorithm 3 shows how each shard processes transactions in Q. To ensure that transactions are executed in order of submission, BFT processes transactions in consecutive round ℓ selected from Q in order. Each round BFT_ℓ picks at most b transactions from the *top* of Q as input, and outputs committed transactions (denoted as TXs_ℓ) (Lines 1-5). For each committed SP-tx$[id]$, if the corresponding request $\mathsf{req}[id]$ is intra-shard, honest parties record SP-tx$[id]$ in the current shard ledger $S_{\mathsf{in}}.\mathsf{log}$, update the output to $\mathsf{UTXO}_{\mathsf{in}}$, and finalize the request processing by responding to the client (Lines 6-10). Otherwise, cross-shard $\mathsf{req}[id]$ is put into cID_ℓ and continues being processed by the following steps.

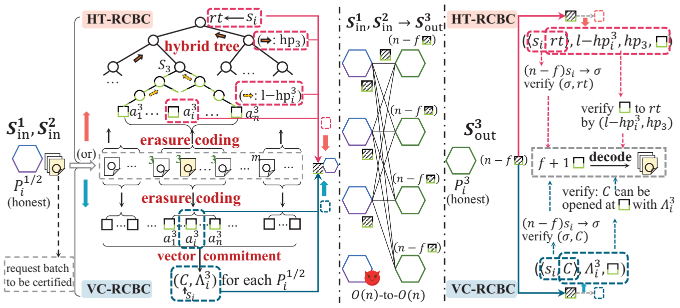
Remark: Algorithm 3 actually represents the invocation process of BFT and does not exclusively belong to a specific step. However, processing different types of transactions triggers different steps.

Algorithm 3 Intra-shard workflow and state update

```
    Let $\mathsf{BFT}_\ell$ execute in consecutive round number $\ell$.
    Initial processed cross-shard request ID set $\mathsf{cID}_\ell = \varnothing$ before $\mathsf{BFT}_\ell$.
    ▶ As a party $P_i$ in shard $S_c$
 1: for $\mathsf{Q} \neq \varnothing$ do
 2:   take out at most $b$ transactions $\{tx\}_b$ from the top of Q   ▷ in order
 3:   execute $\mathsf{BFT}_\ell(\{tx\}_b)$ with function $\mathsf{TxVerify}(tx)$   ▷ SP, FH, BK transactions verification is described in Algorithm 2, 6, 9, respectively
 4:   output transaction set $\mathsf{TXs}_\ell$ committed by $\mathsf{BFT}_\ell(\{tx\}_b)$
 5:   set $\ell \leftarrow \ell + 1$
 6: upon receiving committed $\mathsf{TXs}_\ell$ do   ▷ state update
 7:   for each $tx_i \in \mathsf{TXs}_\ell$ do
 8:     if $tx_i.\mathsf{type} = \text{SP}$ then   ▷ Step 2 and Algorithm 2
 9:       update $\mathsf{UTXO} \leftarrow \mathsf{UTXO} \backslash I[tx_i].\mathsf{utxo}$ if $I[tx_i].\mathsf{utxo} \in \mathsf{UTXO}$
10:       update $\mathsf{log} \leftarrow \mathsf{log} \parallel tx_i$ and $\mathsf{UTXO} \leftarrow \mathsf{UTXO} \cup tx_i.\mathbf{O}$ if $tx_i$ is for an intra-shard request
11:       set $\mathsf{cID}_\ell \leftarrow \mathsf{cID}_\ell \cup tx_i.id$ if $tx_i$ is for a cross-shard request
12:     if $tx_i.\mathsf{type} = \text{FH}$ then   ▷ Step 5 and Algorithm 6
13:       update $\mathsf{log} \leftarrow \mathsf{log} \parallel tx_i$ and $\mathsf{UTXO} \leftarrow \mathsf{UTXO} \cup tx_i.\mathbf{O}$
14:     if $tx_i.\mathsf{type} = \text{BK}$ then   ▷ Step 4# and Algorithm 9
15:       update $\mathsf{UTXO} \leftarrow \mathsf{UTXO} \cup tx_i.\mathbf{O}$
16:   if $\mathsf{cID}_\ell \neq \varnothing$ then
17:     execute reliable cross-shard batch certification (Algorithm 4 or 5)
```

Step ❸ : Cross-shard batch certification (from input shard to output shard).

As cross-shard inputs, SP-txs are expended to other shards, and parties in this input shard inform these expenditures to their corresponding output shards with certificates. As Kronos provides genericity where the applied environment is uncertain, it adopts the most secure cross-shard communication paradigm, $\mathcal{O}(n)$-*to*-$\mathcal{O}(n)$, ensuring reliability without any extra assumption. Kronos provides two methods for batch certification (as shown in Fig. 4), and utilizes erasure coding to decrease the message size of each party.



HT-RCBC
hybrid tree
erasure coding
vector commitment
VC-RCBC
request batch to be certified
for each $P_i^{1/2}$
$S^1_{\text{in}}, S^2_{\text{in}}$
$P_i^{1/2}$ (honest)
$S^1_{\text{in}}, S^2_{\text{in}} \to S^3_{\text{out}}$
$(n-f)$
$\mathcal{O}(n)$-to-$\mathcal{O}(n)$
S^3_{out}
P_i^3 (honest)
$(n-f)s_i \to \sigma$
verify (σ, rt)
verify □ to rt by $(l\text{–}hp_i^3, hp_3)$
$f+1$ □ decode
verify: C can be opened at □ with Λ_i^3
verify (σ, C)

Fig. 4. Cross-shard batch certification.

Method 1: Hybrid tree + erasure coding (Algorithm 4). This method relies on *Merkle tree* technology for efficient batch certification. The tree structure is shown in Fig. 4. Processed requests are categorized by their output shards, and a hash of the requests with the same output shard serves as the leaf node value corresponding to this shard. All leaf nodes are sorted in lexicographic order of the corresponding shard ID and constitute the upper batch tree with root rt. Each leaf node has a *hash path* (denoted as hp) leading to rt, so the output shard can verify the leaf validity by recomputing a root value with the leaf node and corresponding hp, and comparing to the received one. Each leaf node value is computed from a batch of request IDs with a verbose length of $\mathcal{O}(b\lambda)$, so a robust broadcast leading to $CS\text{-}\omega = \mathcal{O}(n^2\lambda(b + \log m))$ between two shards is overburden. Kronos employs erasure coding to decrease the heavy communication overhead.

Input shard operations (Algorithm 4, Lines 1-7): Each party P_i in S_{in} encodes the request IDs sent to S_{out} to n blocks $\{a_i\}_n$. P_i is only responsible for sending a_i (with a length of $\mathcal{O}(\lambda)$) to parties in S_{out} (Lines 1-2). To prevent Byzantine parties from sending fake code blocks, honest parties construct another *lower code tree* with $\{a_i\}_n$ and send their code blocks along with the corresponding hash path and root. Hence, parties in S_{out} can verify the code validity and reconstruct the request IDs upon receiving blocks from $n - f$ parties in the current shard (Line 3). Each party also constructs a Merkle tree, which can be appropriately grafted onto lower code trees of every output shard, constructing a *hybrid tree*. The roots of each lower code tree serve as the leaf nodes of the higher certification tree (Line 4). By introducing this hybrid tree structure, each party P_i only sends $m^i_{\text{BF}} = (\langle \mathsf{rt}, s^{\text{BF}}_i\rangle, a_i, \mathsf{l\text{-}hp}_i, \mathsf{hp_{out}})$ to parties in S_{out}, where s^{BF}_i is the signature share of rt using P_i private key sk^{T}_i, and $\mathsf{l\text{-}hp}_i$ is the hash path from block a_i to the lower tree root (Lines 5-7). Compared with rough broadcast, this method reduces the communication overhead to $\mathcal{O}(n^2\lambda\log m)$ for b transactions.

Output shard operations (Algorithm 4, Lines 8-18): Lines 8-18 show how honest parties in output shard S_{out} learn and verify the processed requests. When an honest party P_i in shard S_{out} receives BUFFER-MESSAGE m_{BF} from P_j in S_{in} within a code block a_j, he verifies the block's validity, i.e., check that a_j can reach rt through hash path $(\mathsf{l\text{-}hp}_j, \mathsf{hp_{out}})$ (Lines 8-10). Upon receiving $n - f$ verified message m_{BF}, honest party P_i combines and verifies the threshold signature using group public key of S_{in} (or verifies the BFT proof when employing determined BFT as remarked below) (Lines 11-12). While the Merkle tree carries hash collision risk, P_i verifies the received blocks collectively by interpolating other non-received ones and reconstructing the tree. Only when the recomputed root value matches the received one can the blocks be used for reliable retrieval (Lines 13-16, 18). 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 $\mathsf{buffer_{out}}$ and awaits integral inputs (Line 17).

Method 2: Vector commitment + erasure coding (Algorithm 5). Method 1 certifies a batch of requests with a message size irrelevant to the specific request number of $\mathcal{O}(b)$, while still maintaining overhead sublinear to the shard number m. We propose another method that utilizes *vector commitment* technology, maintaining a concise message size regardless of either batch size or shard number.

Input shard operations (Algorithm 5, Lines 1-8): Similar to Method 1, parties divide processed requests and encode $\{a^c\}_n$ for each output shard S_c (Lines 1-2). Overall code blocks constitute a $\mathcal{O}(mn)$-sized vector $\boldsymbol{vec} = \big((a^1_1, \cdots, a^1_n), \cdots, (a^m_1, \cdots, a^m_n)\big)$, which is then committed through VecCom with a succinct-length commitment C (Lines

Algorithm 4 Hybrid-tree-based Reliable cross-shard batch certification (HT-RCBC)

Let clD_c link IDs where corresponding spend-transactions are committed to be spent to S_c by the latest round BFT

```
    ▶ As a party $P_i$ in input shard $S_{\mathsf{in}}$ (Step ❸):
 1: for each $(c \in [m]) \wedge (\mathsf{clD}_c \neq \bot)$ do ▷ requests from $S_c$ are processed
 2:     encode $\{a^c_i\}_n \leftarrow \mathsf{ECEnc}(\varphi_c, \mathsf{clD}_c, n, n-2f)$
 3:     compute $(\mathsf{l\text{-}tree}_c, \mathsf{l\text{-}rt}_c, \{\mathsf{l\text{-}hp}_c\}) \leftarrow \mathsf{TreeCon}(\{a^c_i\}_n)$ ▷ lower code tree for coding examination
 4: compute $(\mathsf{tree}, \mathsf{rt}, \{\mathsf{hp}\}) \leftarrow \mathsf{TreeCon}(\{\mathsf{l\text{-}rt}_c\}_{c\in[m]})$ ▷ hybrid higher tree for batch certification, with a maximum of $m$ leaves $\mathsf{l\text{-}rt}_c$
 5: sign $s^{\text{BF}}_i \leftarrow \mathsf{ShareSig}(sk^{\mathsf{T}}_i, \mathsf{rt})$
 6: for each shard $S_c$ with a leaf $\mathsf{l\text{-}rt}_c$ do
 7:     send $m^i_{\text{BF}} = (\langle \mathsf{rt}, s^{\text{BF}}_i\rangle, a_i, \mathsf{l\text{-}hp}_i, \mathsf{hp}_c)$ to parties in $S_c$.
    ▶ As a party $P_i$ in output shard $S_{\mathsf{out}}$ (Step ❸):
 8: upon receiving $m^j_{\text{BF}}$ from $P_j$ of shard $S_{\mathsf{in}}$ do
 9:     parse $m^j_{\text{BF}} = (\langle \mathsf{rt}, s^{\text{BF}}_j\rangle, a_j, \mathsf{l\text{-}hp}_j, \mathsf{hp_{out}})$
10:     check that $(\mathsf{l\text{-}hp}_j, \mathsf{hp_{out}})$ is a valid hash path from leaf $a_j$ to root $\mathsf{rt}$, otherwise discard
11: upon receiving $n-f$ valid $m_{\text{BF}}$ from distinct parties $P_k$ of shard $S_{\mathsf{in}}$ do
12:     compute $\sigma \leftarrow \mathsf{Combine}(\mathsf{rt}, \{(k, s^{\text{BF}}_k)\}_{n-f})$
13:     if $\mathsf{Verify}(\mathsf{rt}, \sigma) = 1$ then
14:         interpolate $a'_j$ from any $n-2f$ $m_{\text{BF}}$ received
15:         recompute Merkle root $\mathsf{rt}'$ and if $\mathsf{rt}' \neq \mathsf{rt}$ then abort
16:         decode $\mathsf{ID_{out}} \leftarrow \mathsf{ECDec}(\varphi_{\mathsf{out}}, \{a_k\}_{n-2f})$
17:         update $\mathsf{buffer} \leftarrow \mathsf{buffer} \bigcup \{(id, I)\}$ where each $id \in \mathsf{ID_{out}}$ and $I$ is managed by $S_{\mathsf{in}}$
18:     otherwise wait for other valid $m_{\text{BF}}$ ▷ there must be $n-f$ $m_{\text{BF}}$ that can pass verification under honest shard configuration
```

3-4). Each a^c_i has a succinct-length proof Λ^c_i, which validates that a^c_i is an element of the vector committed by C. After signing to C with sk^{T}_i, an honest party P_i in S_{in} sends $m^c_{\text{BF}} = (\langle C, s^{\text{BF}}_i\rangle, a^c_i, \Lambda^c_i)$ to parties in S_{out} with a succinct message size of $\mathcal{O}(\lambda)$ (Lines 5-8).

Output shard operations (Algorithm 5, Lines 9-17): Lines 9-17 show the verification and message retrieval operations in S_{out}. As an honest party in S_{out}, P_i verifies the received message $m^j_{\text{BF}} = ((\langle C, s^{\text{BF}}_j\rangle, a_j, \Lambda_j))$ by opening the vector commitment C at a_j with Λ_j. Upon $n - f$ messages with $\mathsf{ComVrf}(\Lambda_j, C, a_j) = 1$, P_i verifies the combined threshold signature (or a BFT proof). The certified request IDs can be reliably retrieved from $n - 2f$ verified blocks a_j once the signature verification is completed (Lines 14-15, 17). Inputs spent by S_{in} for these requests are then stored into $\mathsf{buffer_{out}}$ for future transfer or return (Line 16).

Remark: Both methods require shard parties to sign the commitment (rt of Method 1 or C of Method 2) using their private key share to prove the commitment validity. Careful observation reveals that this operation may seem redundant, as the input expenditures have already been committed by BFT whose proof can naturally prove the commitment'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., [50], [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 (resp. vector) can be constructed *before* BFT execution. By proposing transactions along with the root rt (resp. vector commitment C) in the proposal phase, the BFT proof can guarantee its validity (*batch-proof-with-*

Algorithm 5 Vector-commitment-based Reliable cross-shard batch certification (VC-RCBC)

```
Let $\mathsf{cID}_c$ link IDs where corresponding spend-transactions are committed to be spent to $S_c$ by the latest round BFT
▶ As a party $P_i$ in input shard $S_{\mathsf{in}}$ (Step ❸ ):
1: for each $(c \in [m]) \wedge (\mathsf{cID}_c \neq \perp)$ do
2:     encode $\{a_i^c\}_{[n]} \leftarrow \mathsf{ECEnc}(\varphi_c, ID_c, n, n-2f)$
3: construct vector $\boldsymbol{vec} = \big((a_1^1, \cdots, a_n^1), \cdots, (a_1^m, \cdots, a_n^m)\big)$
4: compute $(C, aux) \leftarrow \mathsf{VecCom}(vec)$   ▷ commit to code block vector
5: sign $s_i^{\text{BF}} \leftarrow \mathsf{ShareSig}(sk_i^{\mathsf{T}}, C)$
6: for each shard $S_c$ with $a_i^c$ do
7:     compute $\Lambda_i^c := \mathsf{ComOpn}(a_i^c, aux)$   ▷ the proof of $a_i^c$ existence in the vector
8:     send $m_{\text{BF}}^c = (\langle C, s_i^{\text{BF}} \rangle, a_i^c, \Lambda_i^c)$ to parties in $S_c$
▶ As a party $P_i$ in output shard $S_{\mathsf{out}}$ (Step ❸ ):
9: upon receiving $m_{\text{BF}}^j$ from $P_j$ of shard $S_{\mathsf{in}}$ do
10:    parse $m_{\text{BF}}^c = (\langle C, s_j^{\text{BF}} \rangle, a_j, \Lambda_j)$
11:    check that $\mathsf{ComVrf}(\Lambda_j, C, a_j) = 1$, otherwise discard.
12: upon receiving $n-f$ valid $m_{\text{BF}}$ from distinct parties of shard $S_{\mathsf{in}}$ do
13:    compute $\sigma \leftarrow \mathsf{Combine}(C, (k, s_k^{\text{BF}}))$
14:    if $\mathsf{Verify}(C, \sigma) = 1$ then
15:        decode $\mathsf{ID_{out}} \leftarrow \mathsf{ECDec}(\varphi_{\mathsf{out}}, \{a_k^{\mathsf{out}}\}_{n-2f})$
16:        update buffer $\leftarrow$ buffer $\bigcup \{(id, I)\}$ where each $id \in \mathsf{ID_{out}}$ and $I$ is managed by $S_{\mathsf{in}}$
17:    else wait for other valid $m_{\text{BF}}$
```

BFT). However, in asynchronous networks, the signing is necessary. According to the FLP "impossibility" [19], asynchronous BFT protocols (e.g., [43], [17], [38], [21], [37], [22]) must run *randomized* subroutines to ensure security, leading to uncertainty about the committed transactions. Therefore, the tree (resp. vector) can only be constructed and signed *after* BFT (*batch-proof-after-BFT*).

Step ❹ : Valid request finalization (by output shard).

In Algorithm 6, when an honest party in S_{out} receives all inputs of request $\mathsf{req}[id]$ (confirming its validity), the party signs to id and multicasts the signature among S_{out} to inform that the request is ready for commitment (Lines 1-3). Once there are $f+1$ valid signatures for id, indicating all honest parties have received integral inputs, $\mathsf{req}[id]$'s inputs in $\mathsf{buffer_{out}}$ become accessible and capable of being transferred by a FINISH-TRANSACTION, $\text{FH-}\mathsf{tx}[id]$. The input of $\text{FH-}\mathsf{tx}[id]$ comprises all inputs of $\mathsf{req}[id]$ stored in $\mathsf{buffer_{out}}$ along with the $(f+1, n)$-threshold signature (serving as t-SIG) (Lines 4-6). Then $\text{FH-}\mathsf{tx}[id]$ is ready to be processed by BFT in Algorithm 3 (Line 7). The verification of $\text{FH-}\mathsf{tx}[id]$ is done by the function TxVerify (Lines 8-11). When a certain round BFT outputs $\text{FH-}\mathsf{tx}[id]$, it is recorded in the output shard ledger and added to $\mathsf{UTXO_{out}}$. Then the stored inputs of $\mathsf{req}[id]$ are removed from buffer, and S_{out} responses to the client to finalize the valid request processing (Algorithm 3, Lines 12-13).

D. Invalid Transaction Rejection of Kronos

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

Algorithm 6 FINISH-TRANSACTION construction (TxCon_{FH})

```
▶ As a party $P_i$ in shard $S_c$: (Step ❹ )
1: for each $\mathsf{req}[id]$ where every $I$ has been stored in buffer do   ▷ $\mathsf{req}[id]$ is valid and ready to be committed
2:     sign ${s_i}^{\text{FH}} := \mathsf{ShareSig}(sk_i^{\mathsf{t}}, \mathsf{H}(\{I\}))$ and multicast $(id, {s_i}^{\text{FH}})$ among $S_c$.   ▷ Apply to execute $\mathsf{req}[id]$.
3:     multicast $(id, {s_i}^{\text{FH}})$ among $S_c$
4:     upon receiving $f+1$ valid $(id, s_j)$ from distinct parties $P_j$ that $\mathsf{ShareVerify}(\mathsf{H}(\{I\}), \{j, s_j\}) = 1$ do
5:         compute $\sigma^{\text{FH}}[id] := \mathsf{Combine}(\mathsf{H}(\{I\}), \{j, s_j\}_{f+1})$   ▷ Serve as the validity proof of $\text{FH-}\mathsf{tx}[id]$
6:         construct $\text{FH-}\mathsf{tx}[id] = (\text{FH}, id, \mathbf{I}, \mathbf{O})$, where $\mathbf{I} = \langle \{I\}, \sigma^{\text{FH}}[id] \rangle$ and $\mathbf{O} = (S_c, pk_c, v_c)$   ▷ transfer received inputs to the payee
7:         append $\text{FH-}\mathsf{tx}[id]$ to Q and wait for BFT to commit it (Algorithm 3)
▶ FH-tx verification: (invoked by Algorithm 3)
8: function TxVerify($tx$)
9:     if $tx$.type = FH then   ▷ $tx$ is a finish-transaction including a combined signature t-SIG = $\sigma^{\text{FH}}$
10:        if $\mathsf{Verify}(\mathsf{H}(\{I_i\}), \sigma^{\text{FH}}) = 1$ where $\mathbf{I} = (\{I_i\}, \sigma^{\text{FH}})$, return 1
11:        otherwise, return 0
```

content-incorrect, where the utxo is non-existent, or sig is invalid. This incorrectness can only be verified by the input shard responsible for managing the input. Secure and comprehensive rejection for an invalid request is achieved as follows:

Step ❷ #: Unavailable input proving (from some input shard to other input and output shards).

As shown in Algorithm 7, upon receiving $\mathsf{req}[id']$, each honest party in S_2 identifies that ${I_2}'$ is unavailable and signs a REJECT-MESSAGE m_{RJ} for $\mathsf{req}[id']$ using their private keys (Lines 1-3). Then, each party multicasts the reject-message to parties in other involved shards (Line 4).

Algorithm 7 Request rejection message transmission (RRMT)

```
Let $I'$ denotes the unavailable input of request $\mathsf{req}[id']$ managed by $S_c$
▶ As a party $P_i$ in shard $S_c$: (Step ❷ #)
1: upon the unavailable input $I'$ of $\mathsf{req}[id']$ do
2:     construct REJECT-MESSAGE $m_{\text{RJ}}[id'] = (\text{RJ}, id', I')$
3:     sign ${s_i}^{\text{RJ}} := \mathsf{ShareSig}(sk_i^{\mathsf{T}}, m_{\text{RJ}}[id'])$
4:     send $\langle m_{\text{RJ}}[id'], {s_i}^{\text{RJ}} \rangle$ to every parties of $\mathsf{req}[id']$ involved shards.
```

Step ❸ #: Invalid request rejection (by input and output shards).

Case 1 (happy path): Abort processing.

Output shard operations (Algorithm 8, Lines 1-5, 7): Upon receiving $n-f$ reject message of $\mathsf{req}[id']$, honest parties in output shard compute threshold signature to verify the message (Lines 1-2). If there are no corresponding inputs of $\mathsf{req}[id']$ in the buffer, then abort the invalid request (Lines 4-5). Afterwards, S_3 responds to the client that $\mathsf{req}[id']$ is rejected (Line 7).

Input shard operations (Algorithm 8, Lines 8-12): Upon receiving $n-f$ reject message for invalid $\mathsf{req}[id']$, honest parties in S_1 verify the threshold signature (Lines 8-10). Then, each party checks whether $\mathsf{req}[id']$ has already been processed. If not (Fig. 3), honest parties quit executing it and eliminate $\text{SP-}\mathsf{tx}(id')$ from Q if it exists (Lines 11-12).

Remark. 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 happy path often occurs with no BFT being "wasted on" invalid requests.

Case 2 (unhappy path): Roll back spent inputs.

Output shard operations (Algorithm 8, Lines 6-7): If there is input in the buffer sent from the input shard, output shard's parties remove the input of invalid $\mathsf{req}[id']$ from buffer_3, and respond to the client that $\mathsf{req}[id']$ is rejected (Lines 6-7).

Input shard operations (Algorithm 9): As Fig. 3 shows, S_2 suffer a terrible latency or adversary delay, and S_1 has spent for $\mathsf{req}[id']$ with SP-$\mathsf{tx}[id']$. For a comprehensive rejection, S_1 returns the spent input to the initial payer through a BACK-TRANSACTION BK-$\mathsf{tx}[id']$ with the received threshold signature as T-SIG, where the utxo is the output of the committed SP-$\mathsf{tx}[id']$ (Lines 1-2). Then, the back-transaction is processed by BFT in Algorithm 3 (Line 3) and verified by the function TxVerify (Lines 4-7). When BK-$\mathsf{tx}[id']$ is output by BFT in S_1, every honest party update UTXO with BK-$\mathsf{tx}[id'].\mathbf{O}$, ensuring a comprehensive rollback (Algorithm 3, Lines 14-15).

Algorithm 8 Rejection and rollback (RRB)

```
    Let req[id'] represent the invalid transaction request
    Let S_in represent input and output shard of invalid req[id'], respectively
    ▶ As a party P_i in shard S_out: (Step ❸ #)
 1: upon receiving $n-f$ $m_{RJ}[id']$ from distinct parties $P_j$ in an input shard of req[id'] do
 2:     compute $\sigma^{RJ}[id'] := \mathsf{Combine}(m_{RJ}[id'], \{(j, s_j)\}_{n-f})$
 3:     if $\mathsf{Verify}(m_{RJ}[id'], \sigma^{RJ}) = 1$ then
 4:         if $\{id', \mathbf{I}\} \notin$ buffer then                ▷ happy path for $S_{out}$
 5:             abort req[id'] processing directly
 6:         otherwise, update buffer ← buffer\{id', I}   ▷ unhappy path for $S_{out}$
 7:         respond to the client that req[id'] commitment failed
    ▶ As a party P_i in shard S_in: (Step ❸ #)
 8: upon receiving $n-f$ $m_{RJ}[id']$ from distinct parties $P_j$ in shard $S_{in}$ do
 9:     compute $\sigma^{RJ}[id'] := \mathsf{Combine}(m_{RJ}[id'], \{(j, s_j)\}_{n-f})$
10:     if $\mathsf{Verify}(m_{RJ}[id'], \sigma^{RJ}) = 1$ then
11:         if SP-tx[id'] ∈ Q_in then                 ▷ happy path for $S_{in}$
12:             remove SP-tx[id'] from Q_in   ▷ abort invalid request processing
13:         otherwise, execute TxCon_BK   ▷ unhappy path for $S_{in}$, TxCon_BK is defined in Algorithm 9
```

Algorithm 9 BACK-TRANSACTION construction ($\mathrm{TxCon}_{\mathrm{BK}}$)

```
    Let req[id'] represent the invalid transaction request
    Let S_in represent input shard of invalid req[id'] and I_in denotes the misspent input
    ▶ As a party P_i in shard S_in: (Step ❸ #)
 1: upon invalid req[id] while S_in has spent I_in for it do
 2:     construct BK-tx = (BK, id, I, O), where I = SP-tx[id].O with T-SIG = $\sigma^{RJ}$. O = $(S_c, pk, v)$ where pk is the initial address of the spent I and v is its value      ▷ get back the misspent inputs
 3:     append BK-tx[id] to Q and wait for BFT to commit it (Algorithm 3)
    ▶ BK-tx verification: (invoked by Algorithm 3)
 4: function TxVerify(tx)
 5:     if tx.type = BK then     ▷ tx is a back-transaction constructed upon receiving a reject-message $m_{RJ}$ with a signature $\sigma^{RJ}$
 6:         if $\mathsf{Verify}(m_{RJ}, \sigma^{RJ}) = 1$ where I.T-SIG = $\sigma^{RJ}$, return 1
 7:         otherwise, return 0
```

V. SECURITY AND COMPLEXITY ANALYSIS

Next, we give security and complexity analysis.

A. 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 each property and its associated guarantees. Please see Appendix B for detailed proofs.

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 $S_c.\mathsf{log}_j$ recorded by any honest party P_j in shard S_c.*

Proof of Theorem 1: 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 $tx_1.\mathbf{I} \cap tx_2.\mathbf{I} \neq \varnothing$.*

Proof of Theorem 2: Consistency is ensured by TxVerify of BFT, cross-shard certification, and threshold buffer management. ■

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

Proof of Theorem 3: The atomicity property is ensured by the output shard waiting for integral inputs before commitment, cross-shard certification, and rollback mechanism achieved with back-transactions. ■

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

Proof of Theorem 4: The liveness property is guaranteed by the introduced submission paradigm, intra-shard BFT liveness, and reliable cross-shard communication. ■

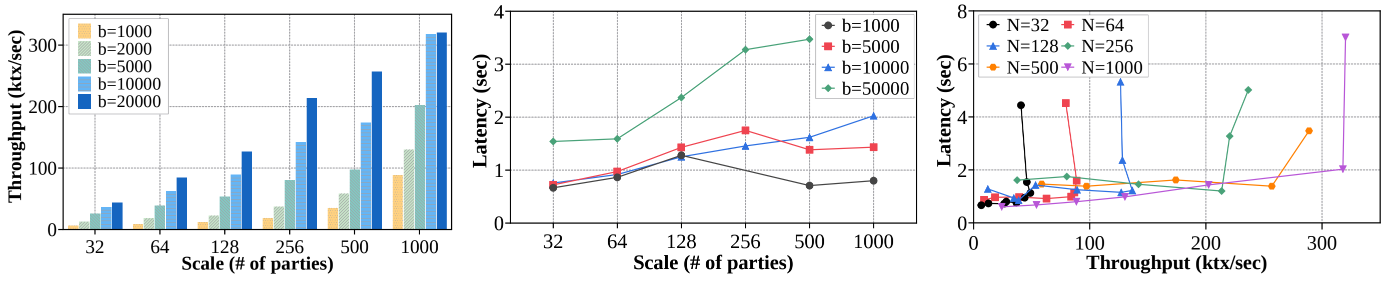
B. Complexity Analysis

In this section, we analyze the intra-shard and cross-shard communication overhead for transaction processing in Kronos. Additionally, we delve into the lower bound of intra-shard overhead while ensuring secure transaction processing. Please refer to Appendix C for detailed proof of Theorem 5 and 6.

Intra-shard communication overhead. As stated in Theorem 5, Kronos processes transactions with optimal intra-shard communication overhead $k\mathcal{B}$.

Theorem 5 (Optimal intra-shard communication overhead)**.** Kronos *commits a k-shard-involved transaction tx through executing BFT protocols k times, realizing the lower bound of intra-shard communication overhead IS-ω= $k\mathcal{B}$, which is optimal for a secure sharding blockchain as per Definition 1.*

Cross-shard communication overhead. Definition 4 describes the reliability property for cross-shard transmission.



(a) Peak throughput (b) Latency (c) Latency-throughput trade-off

Fig. 5. Performance of sKronos in the WAN setting.

Theorem 6 depicts that Kronos realizes a low overhead of $CS\text{-}\omega$ $\mathcal{O}(n^2\lambda)$ where n denotes the shard size and λ is the security parameter.

Definition 4 (Cross-shard message transmission reliability)**.** *For a cross-shard message transmission, reliability means that the message is ensured to be sent by the source shard and guaranteed to be received by at least one honest party in the destination shard.*

Theorem 6 (Cross-shard communication overhead)**.** Kronos *completes a round of cross-shard communication with a overhead of $\mathcal{O}(n^2\lambda)$, and realizes cross-shard message transmission reliability as per Definition 4.*

VI. 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 evaluated aspects include the overall performance of Kronos in realistic wide-area network (Section VI-A), the performance improvement compared to existing sharding protocols (Section VI-B), and whether it is truly generic and scalable in various network models with different BFT protocols (Section VI-C).

Implementation details. We program the implementations of Kronos and 2PC in the same language Python [2]. All libraries and security parameters required in cryptographic implementations are the same. All nodes are assigned into shards, each adopting Speeding Dumbo [22] (an efficient and robust asynchronous BFT protocol) 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 [50]. All hash functions are instantiated using SHA256 [31]. Cross-shard certification is realized on hybrid trees.

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 equipped with 16 vCPUs and 32GB main memory. The performances are evaluated with varying scales at $N = 16, 32$, 64, 128, 256, 500, and 1000 nodes. The proportion of cross-shard transactions varies between 10%, 30%, 50%, 70%, 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.

[2]We plan to submit and opensource our artifacts.

A. Overall Performance of Kronos

Throughput and latency. To evaluate Kronos, 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 adopted BFT (i.e., the number of transactions proposed in each round) from $b = 1\text{k}$ to 200k for evaluation (4 nodes in each shard). This reflects how well Kronos performs in realistic scenarios.

As illustrated in Fig. 5a, sKronos demonstrates scalability, showcasing an increasing throughput as network scales and achieving a peak throughput of 320.2 ktx/sec with $N = 1000$, $b = 20\text{k}$. As the network scales, the optimal batch size for achieving peak throughput decreases. Fig. 5b illustrates sKronos latency across varying network sizes, where latency is measured as the time elapsed between a request entering the waiting queue and its processing completion. The latency remains below 2.03 sec for network scales $N \leq 1000$ and batch sizes $b \leq 10\text{k}$, highlighting the effectiveness of Kronos for latency-critical applications, even at a large scale.

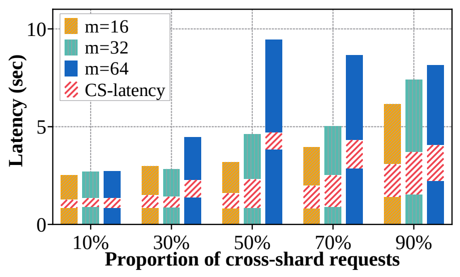
Throughput-latency trade-off. Fig. 5c illustrates the latency-throughput trade-off of sKronos. The latency stays below 1.13 sec, with throughput reaching 136.7 ktx/sec in a medium-scale network ($N = 100$). In large-scale networks ($N = 500$), the latency remains low as the throughput reaches a sizable 256.7 ktx/sec. This trade-off underscores the applicability of Kronos in scenarios requiring both throughput and latency.

B. Performance on Cooperation Across Shards and Comparison with Existing Solutions

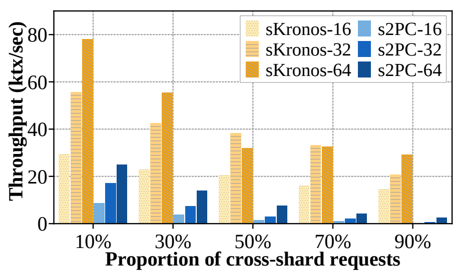
To analyze how well Kronos handles cross-shard requests, we evaluate the specific time cost of each processing step of sKronos. Besides, we compare the throughput and latency of sKronos and s2PC with varying cross-shard transaction proportions.

Cross-shard latency. We evaluate the specific cost of cross-shard request processing by measuring the latency at each step during a cross-shard request processing, as shown in Fig. 6a. 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.

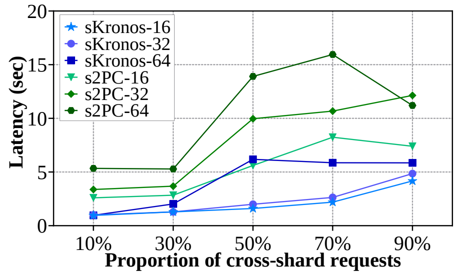
The experimental results reveal the cost of cross-shard cooperation highlighted in red (with the same batch size



(a) sKronos cross-shard latency

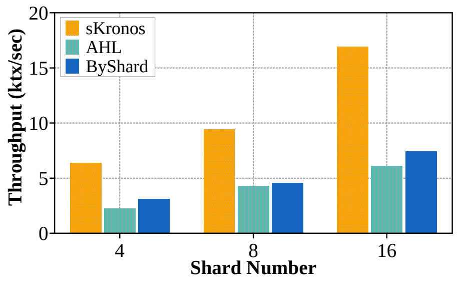


(b) sKronos v.s. s2PC (TPS)

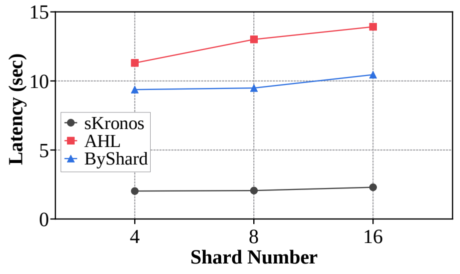


(c) sKronos v.s. s2PC (latency)

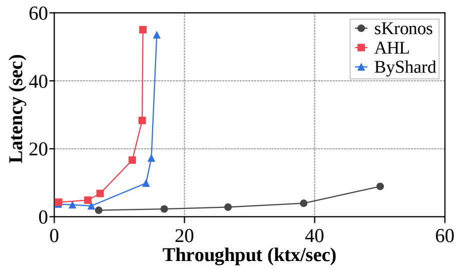
Fig. 6. The efficiency of Kronos (sKronos and s2PC utilize Speeding Dumbo).



(a) sKronos v.s. AHL and ByShard (TPS)



(b) sKronos v.s. AHL and ByShard (latency)



(c) sKronos v.s. AHL and ByShard (trade-off)

Fig. 7. Comparison with other solutions.

$b = 10$k). When the shard number m is 16, and the cross-shard request proportion is 10%, the cross-shard time cost is about 0.43 sec, occupying less than 17% of the total latency. As the cross-shard request proportion increases to 90%, the cost slightly rises but stays below 28% of the total. In a larger-scale system with 64 shards, the impact remains lower than 30% with 90% cross-shard requests, illustrating 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.

Fig. 6b and Fig. 6c depict the throughput and latency of sKronos in comparison to s2PC. Overall, sKronos outperforms s2PC in all cases. Notably, the throughput of sKronos within 64 shards exceeds $2.4\times$ that of s2PC when 10% of requests are cross-shard, approximately $4\times$ when the proportion is 50%, and an impressive $12\times$ when the proportion is 90%! In terms of latency, sKronos incurs at most half the time cost of s2PC (in all the cross-shard proportion cases).

Comparison with other sharding blockchains. We also compare sKronos with state-of-the-art systems, AHL [12] and ByShard [24], in the same network scale (each shard comprises 30 nodes). Fig. 7 demonstrates that the throughput of sKronos exceeds ByShard's by $2.3\times$ and AHL's by $2.7\times$, with a time cost below one-third. Additionally, sKronos exhibits a significant advantage in peak throughput with low latency, making it applicable in both latency-critical and throughput-critical scenarios.

C. Performance on Various BFT

Finally, we substitute the intra-shard consensus with HotStuff to showcase Kronos generality across different systems. Fig. 8 illustrates latency-throughput trade-offs of sKronos, hKronos, and s2PC. hKronos also exhibits high efficiency, achieving a peak throughput of 1.2×10^2 ktx/sec with a low latency of 1.78 sec when $N = 256$. sKronos reaches at least $3\times$ 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.

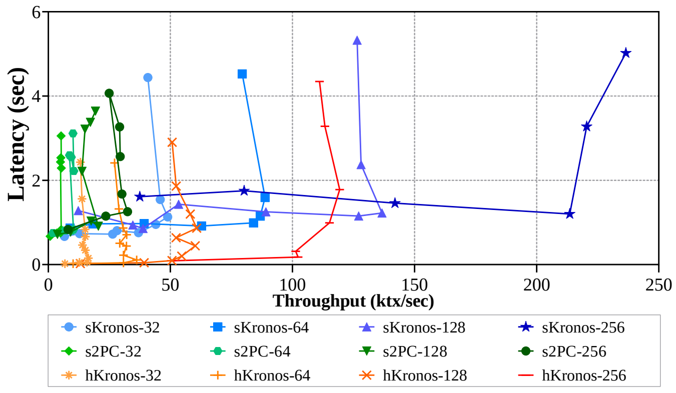


Fig. 8. Latency-throughput trade-off of several consensus.

VII. Conclusion

We present Kronos, the first *generic* sharding blockchain consensus that realizes robust security and optimized overhead even in asynchronous networks. The proposed new sharding consensus pattern realizes atomicity with optimal intra-shard overhead. Two concrete cross-shard batch certification constructions realize reliability with low cross-shard overhead. Implementation results demonstrate Kronos outstanding scalability, surpassing existing solutions, making it suitable for practical applications. In particular, Kronos could be utilized as a universal framework for enhancing the performance and scalability of existing BFT protocols, supporting all network models, including asynchronous ones. Kronos scales the consensus nodes to thousands and increases the throughput by several orders of magnitude, which is unprecedented.

References

[1] I. Abraham, P. Jovanovic, M. Maller *et al.*, "Reaching consensus for asynchronous distributed key generation," in *PODC'21*. ACM, 2021, pp. 363–373.

[2] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin, "Sync hotstuff: Simple and practical synchronous state machine replication," in *SP'20*. IEEE, 2020, pp. 106–118.

[3] M. Al-Bassam, A. Sonnino, S. Bano *et al.*, "Chainspace: A sharded smart contracts platform," in *NDSS'18*. ISOC, 2018.

[4] G. Almashaqbeh and R. Solomon, "Sok: Privacy-preserving computing in the blockchain era," in *EuroS&P'22*. IEEE, 2022, pp. 124–139.

[5] M. J. Amiri, D. Agrawal, and A. El Abbadi, "Sharper: Sharding permissioned blockchains over network clusters," in *SIGMOD'21*. ACM, 2021, pp. 76–88.

[6] Z. Avarikioti, A. Desjardins, L. Kokoris-Kogias, and R. Wattenhofer, "Divide & scale: Formalization and roadmap to robust sharding," in *Structural Information and Communication Complexity*. Springer Nature Switzerland, 2023, pp. 199–245.

[7] F. Benhamouda, S. Halevi, H. Krawczyk *et al.*, "Threshold cryptography as a service (in the multiserver and yoso models)," in *CCS'22*. ACM, 2022, pp. 323–336.

[8] D. Boneh, M. Drijvers, and G. Neven, "Compact multi-signatures for smaller blockchains," in *ASIACRYPT'18*. Springer, 2018, pp. 435–464.

[9] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," *Journal of cryptology*, vol. 17, pp. 297–319, 2004.

[10] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OsDI'99*. USENIX Association, 1999, pp. 173–186.

[11] A. Clement, E. Wong, L. Alvisi, M. Dahlin, M. Marchetti *et al.*, "Making byzantine fault tolerant systems tolerate byzantine faults," in *USENIX Security'09*. The USENIX Association, 2009.

[12] H. Dang, T. T. A. Dinh, D. Loghin *et al.*, "Towards scaling blockchain systems via sharding," in *SIGMOD'19*. ACM, 2019, pp. 123–140.

[13] S. Das, V. Krishnan, and L. Ren, "Efficient cross-shard transaction execution in sharded blockchains," *arXiv preprint arXiv:2007.14521*, 2020.

[14] S. Das, T. Yurek, Z. Xiang *et al.*, "Practical asynchronous distributed key generation," in *SP'22*. IEEE, 2022, pp. 2518–2534.

[15] B. David, B. Magri, C. Matt *et al.*, "Gearbox: Optimal-size shard committees by leveraging the safety-liveness dichotomy," in *CCS'22*. ACM, 2022, pp. 683–696.

[16] S. Duan, S. Peisert, and K. N. Levitt, "hbft: speculative byzantine fault tolerance with minimum cost," *IEEE Trans. Dependable Secur. Comput.*, vol. 12, no. 1, pp. 58–70, 2014.

[17] S. Duan, M. K. Reiter, and H. Zhang, "Beat: Asynchronous bft made practical," in *CCS'18*. ACM, 2018, pp. 2028–2041.

[18] S. Duan, X. Wang, and H. Zhang, "Fin: Practical signature-free asynchronous common subset in constant time," in *CCS'23*. ACM, 2023, pp. 815–829.

[19] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *JACM*, vol. 32, no. 2, pp. 374–382, 1985.

[20] M. Fitzi, P. Ga, A. Kiayias, and A. Russell, "Parallel chains: Improving throughput and latency of blockchain protocols via parallel composition," 2018, https://eprint.iacr.org/2018/1119.pdf.

[21] Y. Gao, Y. Lu, Z. Lu *et al.*, "Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency," in *CCS'22*. ACM, 2022, pp. 1187–1201.

[22] B. Guo, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Speeding dumbo: Pushing asynchronous BFT closer to practice," in *NDSS'22*. ISOC, 2022.

[23] B. Guo, Z. Lu, Q. Tang *et al.*, "Dumbo: Faster asynchronous bft protocols," in *CCS'20*. ACM, 2020, pp. 803–818.

[24] J. Hellings and M. Sadoghi, "Byshard: sharding in a byzantine environment," *VLDB J.*, vol. 32, no. 6, pp. 1343–1367, 2023.

[25] Z. Hong, S. Guo, and P. Li, "Scaling blockchain via layered sharding," *IEEE J. Sel. Areas Commun.*, vol. 40, no. 12, pp. 3575–3588, 2022.

[26] Z. Hong, S. Guo, P. Li, and W. Chen, "Pyramid: A layered sharding blockchain system," in *INFOCOM'21*. IEEE, 2021, pp. 1–10.

[27] H. Huang, X. Peng, J. Zhan *et al.*, "Brokerchain: A cross-shard blockchain protocol for account/balance-based state sharding," in *INFOCOM'22*. IEEE, 2022, pp. 1968–1977.

[28] T. Huynh-The, T. R. Gadekallu, W. Wang *et al.*, "Blockchain for the metaverse: A review," *Futur. Gener. Comp. Syst.*, 2023.

[29] S. Jiang, J. Cao, C. L. Tung, Y. Wang, and S. Wang, "Sharon: Secure and efficient cross-shard transaction processing via shard rotation," 2024.

[30] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel, "Cheapbft: Resource-efficient byzantine fault tolerance," in *CCS'12*. ACM, 2012, pp. 295–308.

[31] A. K. Kasgar, J. Agrawal, and S. Shahu, "New modified 256-bit md 5 algorithm with sha compression function," *Int. J. Comput. Appl. Technol.*, vol. 42, no. 12, 2012.

[32] E. Kokoris-Kogias, P. Jovanovic, L. Gasser *et al.*, "Omniledger: A secure, scale-out, decentralized ledger via sharding," in *SP'18*. IEEE, 2018, pp. 583–598.

[33] Y. Lin, Z. Gao, H. Du *et al.*, "A unified blockchain-semantic framework for wireless edge intelligence enabled web 3.0," *IEEE Wirel. Commun.*, vol. 31, no. 2, pp. 126–133, 2024.

[34] Y. Liu, J. Liu, M. A. V. Salles *et al.*, "Building blocks of sharding blockchain systems: Concepts, approaches, and open problems," *Comput. Sci. Rev.*, vol. 46, p. 100513, 2022.

[35] Y. Liu, J. Liu, Q. Wu *et al.*, "SSHC: A secure and scalable hybrid consensus protocol for sharding blockchains with a formal security framework," *IEEE Trans. Dependable Secur. Comput.*, vol. 19, no. 3, pp. 2070–2088, 2020.

[36] Y. Liu, X. Xing, H. Cheng *et al.*, "A flexible sharding blockchain protocol based on cross-shard byzantine fault tolerance," *IEEE Trans. Inf. Forensics Secur.*, vol. 18, pp. 2276–2291, 2023.

[37] Y. Lu, Z. Lu, and Q. Tang, "Bolt-dumbo transformer: Asynchronous consensus as fast as the pipelined bft," in *CCS'22*. ACM, 2022, pp. 2159–2173.

[38] Y. Lu, Z. Lu, Q. Tang, and G. Wang, "Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited," in *PODC'20*. ACM, 2020, pp. 129–138.

[39] M. Luby, M. Mitzenmacher, M. A. Shokrollahi, and D. A. Spielman, "Efficient erasure correcting codes," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 569–584, 2001.

[40] L. Luu, V. Narayanan, C. Zheng *et al.*, "A secure sharding protocol for open blockchains," in *CCS'16*. ACM, 2016, pp. 17–30.

[41] Y. Ma, J. Woods, S. Angel, A. Polychroniadou, and T. Rabin, "Flamingo: Multi-round single-server secure aggregation with applications to private federated learning," in *SP'23*. IEEE, 2023, pp. 477–496.

[42] D. Maram, H. Malvai, F. Zhang, N. Jean-Louis, A. Frolov, T. Kell, T. Lobban, C. Moy, A. Juels, and A. Miller, "Candid: Can-do decentralized identity with legacy compatibility, sybil-resistance, and accountability," in *SP'21*. IEEE, 2021, pp. 1348–1366.

[43] A. Miller, Y. Xia, K. Croman *et al.*, "The honey badger of bft protocols," in *CCS'16*. ACM, 2016, pp. 31–42.

[44] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized business review*, p. 21260, 2008.

[45] R. Neiheiser, M. Matos, and L. E. T. Rodrigues, "Kauri: Scalable BFT consensus with pipelined tree-based dissemination and aggregation," in *SOSP'21*. ACM, 2021, pp. 35–48.

[46] S. Srinivasan, A. Chepurnoy, C. Papamanthou, A. Tomescu, and Y. Zhang, "Hyperproofs: Aggregating and maintaining proofs in vector commitments," in *USENIX Security'22*. USENIX Association, 2022, pp. 3001–3018.

[47] R. Vassantlal, E. Alchieri, B. Ferreira, and A. Bessani, "Cobra: Dynamic proactive secret sharing for confidential bft services," in *SP'22*. IEEE, 2022, pp. 1335–1353.

[48] J. Wang and H. Wang, "Monoxide: Scale out blockchains with asynchronous consensus zones," in *NSDI'19*, vol. 2019, 2019, pp. 95–112.

[49] Y. Xu, J. Zheng, B. Düdder, T. Slaats, and Y. Zhou, "A two-layer blockchain sharding protocol leveraging safety and liveness for enhanced performance," in *NDSS'24*. ISOC, 2024.

[50] M. Yin, D. Malkhi, M. K. Reiter *et al.*, "Hotstuff: Bft consensus with linearity and responsiveness," in *PODC'19*. ACM, 2019, pp. 347–356.

[51] T. Yurek, Z. Xiang, Y. Xia, and A. Miller, "Long live the honey badger: Robust asynchronous DPSS and its applications," in *USENIX Security'23*. USENIX Association, 2023, pp. 5413–5430.

[52] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," in *CCS'18*. ACM, 2018, pp. 931–948.

[53] J. Zhang, W. Chen, S. Luo, T. Gong *et al.*, "Front-running attack in sharded blockchains and fair cross-shard consensus," in *NDSS'24*. ISOC, 2024.

[54] M. Zhang, J. Li, Z. Chen *et al.*, "An efficient and robust committee structure for sharding blockchain," *IEEE Trans. Cloud Comput.*, vol. 11, no. 3, pp. 2562–2574, 2023.

[55] P. Zheng, Q. Xu, Z. Zheng *et al.*, "Meepo: Multiple execution environments per organization in sharded consortium blockchain," *IEEE J. Sel. Areas Commun.*, vol. 40, no. 12, pp. 3562–3574, 2022.

Appendix 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.

Appendix 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 $S_c.\mathsf{log}_j$ recorded by any honest party P_j in shard S_c.*

Proof of Theorem 1: 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 is configured to be honest, with the proportion of Byzantine parties being less than $n/3$ in partial synchronous and asynchronous networks, and $n/2$ in synchronous networks, ensuring BFT protocols' safety and liveness successfully.

In a given shard S_c, a secure BFT protocol with an external function for transaction verification is employed. If an honest party outputs transaction tx in a committed set TXs_ℓ in a certain round ℓ, then any honest 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. 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 $tx_1 \neq tx_2$ and $tx_1.\mathbf{I} \cap tx_2.\mathbf{I} \neq \varnothing$.*

Proof of Theorem 2: We prove it by contradiction. Suppose that there exist two conflicting transactions tx_1 and tx_2 where $tx_1.\mathbf{I} \cap tx_2.\mathbf{I} = \overline{I}$ within $\overline{\mathsf{utxo}}$. By 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 SP-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 before recording, tx_1 and tx_2 cannot be committed in the same round BFT. In each BFT round, TxVerify examines SP-txs with the signature sig and the utxo existence in UTXO, so the conflict arises only if $\overline{\mathsf{utxo}}$ 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 $\overline{\mathsf{utxo}}$ 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, SP-tx only spend utxo managed by the current shard and any utxo only belongs to a single shard, then conflicting SP-txs in different shards are impossible. Therefore, there is no conflict between SP-txs.

Consistency among spend and finish-transactions. Suppose that tx_1 and tx_2 are SP-tx and FH-tx, respectively. According to TxVerify, a SP-tx is verified valid only if SP-tx.$\mathbf{I} \subseteq$ UTXO. However, FH-tx.$\mathbf{I}$ is from buffer. If $tx_1.\mathbf{I} \cap tx_2.\mathbf{I} = \overline{I}$, there must be a round where $\overline{\mathsf{utxo}} \in$ UTXO and $\overline{\mathsf{utxo}} \in$ 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_{BF}. No transaction output belongs to UTXO and buffer simultaneously. Therefore, there is no conflict between SP-tx and FH-tx.

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

Consistency among finish-transactions. If there are two conflict

FH-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 FH-tx in any shard. FH-tx inputs are stored in buffer with their corresponding ids, thus one input can only be transferred by a FH-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 SP-txs. While it is proven that there is no conflict between SP-txs, so there is no conflict between FH-txs.

Consistency among finish and back-transactions. FH-tx and BK-tx inputs are both from buffer where the inputs are stored with corresponding id and are removed once the transaction is recorded. In case the ids of conflict FH-tx tx_1 and BK-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 SP-txs. So there is no conflict between FH-tx and BK-tx with different id. In case the ids of conflict FH-tx tx_1 and BK-tx tx_2 are the same, FH-tx$[id]$ is signed only if at least $n-f$ parties have received all inputs of req$[id]$ and stored them in buffer, while BK-tx$[id]$ is committed after receiving $m_{\text{RJ}}[id]$ 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, FH-tx$[id]$ and BK-tx$[id]$ for the same req$[id]$ do not exist. Therefore, there is no conflict between FH-tx and BK-tx.

Consistency among back-transactions. Similar to FH-tx, each id corresponds to only one BK-tx. If BK-tx$[id]$ and BK-tx$[id']$ are in conflict, there must be some stored inputs from two conflicting SP-txs. While it is proven that there is no conflict between SP-txs, so there is no conflict between BK-txs. ■

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 final fund transfer if it is invalid.*

Proof of Theorem 3: 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 SP-tx$[\gamma]$ to spend the required inputs. The valid transaction SP-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 reliably delivered by cross-shard message transmission to the output shard with batch certification, 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 $f+1$ valid signatures and transferred to the payee's address, finalizing the execution. Therefore, a valid transaction request is executed by all corresponding 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, when managing an unavailable input, verifies req$[\gamma]$ after receiving it, thereby preventing the execution of req$[\gamma]$ by any transaction. Instead, it generates 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 SP-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 BK-tx$[\gamma]$ to pay back the spent input to initial address with the received signature σ^{RJ} in m_{RJ} as T-SIG.

An honest shard cannot reject req$[\gamma]$ using a $(n-f,n)$-threshold signature σ^{RJ} in a reject-message while simultaneously spending on req$[\gamma]$, certified by signatures from a majority of shard members in a buffer-message. This is because at most f Byzantine nodes might both vote to spend the transaction and sign unavailability for the same request equivocally. Therefore, when a rejection message is received, the output shard's buffer can never accumulate a sufficient number of valid inputs for req$[\gamma]$. Afterwards, each honest party in the output shard empties req$[\gamma]$'s inputs stored in buffer. Therefore, invalid req$[\gamma]$ is rejected by each involved shard finally with no state change. ■

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

Proof of Theorem 4: The liveness property is guaranteed by the submission paradigm and intra-shard BFT liveness.

Liveness in valid request processing. If req$[\gamma]$ is valid, the output shard forwards it to all involved shards. Each honest input shard verifies the request and creates a spend-transaction SP-tx$[\gamma]$ to spend available inputs. As shards select transactions for BFT from the waiting queue Q in order, SP-tx$[\gamma]$ is selected and proposed within a limited time. Due to the liveness of BFT, SP-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 $\kappa = \kappa_{\text{BFT}}$. 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} by a cross-shard communication round. The buffer eventually stores all inputs of the valid request, and every honest party in the output shard signs to the validity of req$[\gamma]$ via a round of intra-shard communication. The finish-transaction FH-tx$[\gamma]$ is constructed after collecting $f+1$ valid signatures from distinct parties and then committed by BFT by κ_{BFT} rounds of intra-shard communication. Therefore, the liveness parameter holds that $\kappa = 2\kappa_{\text{BFT}} + 2$.

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 cross-shard communication for invalidity transfer 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 BK-tx$[\gamma]$ within another κ_{BFT} rounds of intra-shard communication. In total, the worst-case liveness parameter is $\kappa = 2\kappa_{\text{BFT}} + 1$.

In summary, a submitted request must be processed within κ rounds of intra-shard or cross-shard communication, where $\kappa \leq 2\kappa_{\text{BFT}} + 2$. ■

Appendix C
Complexity Analysis

Theorem 5 (Optimal intra-shard communication overhead)**.** Kronos *commits a k-shard-involved transaction tx through executing BFT protocols k times totally, realizing the lower bound of intra-shard communication overhead IS-ω= kℬ, which is optimal for a secure sharding blockchain as per Definition 1.*

Proof of Theorem 5: A cross-shard transaction is executed with a state update to the UTXO/accounts and 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 $k-1$ 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, $k\mathcal{B}$ is the lower bound of $IS\text{-}\omega$ 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, where the $IS\text{-}\omega$ is equal to the lowest bound $k\mathcal{B}$. 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 proof. In this case, Kronos processes invalid transactions with the optimal overhead $IS\text{-}\omega = 0$ without requiring any BFT execution.

To ensure atomicity, Kronos allows rollback by input shards already spent for an invalid transaction request. The inputs are returned through another round of BFT, leading to 2 rounds of BFT inside the shard for rejection. Consider an invalid transaction with x inputs where x' of them are unavailable. The unhappy path $IS\text{-}\omega$ for it is $2(x-x')\mathcal{B}$. In Kronos, this worst situation rarely occurs due to responsive rejection. Any available input gets spent only after the SP-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 messages for rejection are constructed and sent by honest parties responsively on receiving the invalid request, making it faster than the spending process. Therefore, Kronos achieves optimistic $IS\text{-}\omega$ requiring no BFT for invalid transactions in most instances, and there are at most $2(x-x')$ rounds of BFT protocol executed for it without any extra storage or computation overhead even in the unhappy path. ■

Theorem 6 (Cross-shard communication overhead)**.** Kronos *completes a round of cross-shard communication with a overhead of $\mathcal{O}(n^2\lambda)$, and realizes cross-shard message transmission reliability as per Definition 4.*

Proof of Theorem 6: Communication overhead can be reduced by either decreasing each message size or by minimizing the exchanged message number. When using the $\mathcal{O}(n)$-*to*-$\mathcal{O}(n)$ paradigm for reliability, Kronos delves into reducing message size from the two parts, $\mathcal{O}(b\lambda)$-sized request ID and $\mathcal{O}(b\lambda)$-sized commitment. The former is decreased through erasure coding, where $\mathcal{O}(b\lambda)$-sized IDs are encoded to n concise code blocks where each party only sends one $\mathcal{O}(\frac{b\lambda}{n})$-sized block. For the code block certification and commitment compression, Kronos offers two methods adopting Merkle trees and vector commitment (in HT-RCBC and VC-RCBC), respectively. When utilizing the Merkle tree technology, requests are constructed to a batch tree with m leaf nodes, and code blocks are constructed to the code tree with n leaf nodes. Each party sends the code block along with a hash path with the size of $\mathcal{O}((\log m + \log n)\lambda)$. Therefore, the communication overhead of Kronos using HT-RCBC holds that $CS\text{-}\omega_{\text{HT}} = n^2(\mathcal{O}(\frac{b\lambda}{n}) + \mathcal{O}((\log m + \log n)\lambda)) = \mathcal{O}(n(b + n\log m + n\log n)\lambda)$. When the system scales such that both $n\log m$ and $n\log n$ are smaller than the BFT batch size b (e.g., $m=50$, $n=64$, and $b=10^4$), the overhead reaches $\mathcal{O}(nb\lambda)$ that is linear to the batch size.

When utilizing vector commitment where the commitment value and proofs at each position are both succinct $\mathcal{O}(\lambda)$ size, each message part maintains concise and the communication overhead $CS\text{-}\omega_{\text{VC}} = n^2\mathcal{O}(\frac{b\lambda}{n} + \lambda) = \mathcal{O}(nb\lambda)$ regardless of the specific network scale. Therefore, Kronos realizes a low overhead of $\mathcal{O}(nb\lambda)$.

Notice that when using $\mathcal{O}(1)$-to-$\mathcal{O}(1)$ message delivery, since the sender or receiver may be malicious, it is necessary to employ a time parameter Δ (for synchronous networks) or a timeout (for partially synchronous networks) to replace the message sender/receiver until honest nodes are found. Consequently, achieving reliability in this manner also incurs a cross-shard communication complexity of $\mathcal{O}(nb\lambda)$. ■

Appendix D
Other Optional Modules

Optional cross-shard communication paradigm. While our analysis indicates that operating cross-shard communication with an $\mathcal{O}(n)$-*to*-$\mathcal{O}(n)$ paradigm ensures generic security, it does not imply the total elimination of other paradigms (e.g., *leader-to-leader*, $\mathcal{O}(1)$-*to*-$\mathcal{O}(1)$ for $\mathcal{O}(n)$ rounds). As long as the dependency is carefully ensured by design (e.g., through a cross-shard view-change mechanism in Algorithm 10 ensuring leader honesty), the reliable cross-shard batch certification of Kronos can be achieved. Clients first submit requests to all parties in the output shard. In the $\mathcal{O}(1)$-*to*-$\mathcal{O}(1)$ mode, each party sends the constructed BUFFER-MESSAGE or REJECT-MESSAGE with a signature share to the shard leader. The leader combines the threshold signature and sends the combined message to leaders of destination shards. In a happy path where the leaders are honest, the message can be reliably received by parties in the destination shard. In case of an unhappy path where Byzantine leaders exist, honest shard members can detect this error through timers and execute a view-change to change the potentially malicious leader with a timeout proof (utilizing a quorum of signatures).

Algorithm 10 Cross-shard view change

```
**Each** party in output shard starts a timer $\tau$ for cross-shard request $\mathsf{req}[id]$ after delivering it to leaders of its input shards.
    ▶ As a party $P_i$ in output shard $S_{\mathsf{out}}$:
 1: if ($\tau$ ends)$\wedge$ (neither reject-message $m_{\text{RJ}}$ nor buffer-message $m_{\text{BF}}$ with valid threshold signature is received from some input shard $S_{\mathsf{in}}$) then
 2:     initiate a view-change to change the leader of $S_{\mathsf{out}}$
 3:     send $\mathsf{req}[id]$ to all parties in $S_{\mathsf{in}}$ along with a proof of the timeout
    ▶ As a party $P_i$ in $S_{\mathsf{in}}$:
 4: upon receiving $\mathsf{req}[id]$ with a proof of timeout from parties in $S_{\mathsf{out}}$ do
 5:     initiate a view-change to change the leader of $S_{\mathsf{in}}$
 6:     if $\mathsf{req}[id]$ has been processed then
 7:         send the corresponding BUFFER-MESSAGE to parties in $S_{\mathsf{out}}$
 8:     if $\mathsf{req}[id]$ has been signed as invalid then
 9:         send the corresponding REJECT-MESSAGE to parties in $S_{\mathsf{out}}$
10:     otherwise, wait for the new leader to propose $\mathsf{req}[id]$
```

Realizing buffer using multi-signature scheme. The primary role of buffer within output shards of cross-shard requests is to manage the received input by all shard members securely. In this regard, Kronos employs threshold signatures, where the threshold of $f+1$ ensures trusted management. Additionally, a multi-signature scheme is also viable. Each shard can designate a group address (e.g., the product of the multi-signature public keys of all nodes within a shard) specially used for storing inputs from other shards, and only a multi-signature combined from a quorum of $f+1$ valid signatures can access and operate on this address. In this way, the DKG for $(f+1, n)$-threshold signature becomes unnecessary (if the intra-shard BFT does not demand it) but instead necessitates a mechanism for generating or certifying keys for the multi-signature.

Secure buffer management during shard reconfiguration. In the implementation of buffer, there is a minor consideration related to shard reconfiguration. To prevent specific shards from falling under complete control due to a gradual corruption of more than $1/3$ of the 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" the buffer by acquiring comprehensive information about each input. This allows new members to transfer or refund the inputs to complete the transaction processing. A DKG protocol 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.

Attentive observation reveals that this process could be streamlined with the implementation of *dynamic-committee proactive secret sharing* (DPSS) [47], [51]. 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 shard reconfiguration process ensures a fixed share public key and buffer address, and old shard members (might be corrupted) cannot execute any operations on the buffer.