Artifact Evaluated NDSS Available Functional

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§
*Beihang University, Email: {liuyizhong, liuandi, zhuochengPan, liujianwei, sbian}@buaa.edu.cn
†Institute of Software, Chinese Academy of Sciences, Email: luyuan@iscas.ac.cn
‡Xi'an Jiaotong University, Email: yynnliyinuo@stu.xjtu.edu.cn
§University of Padua, Email: conti@math.unipd.it

***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 blockchain 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 blockchain 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 while maintaining *optimal intra-shard overhead*. Efficient rejection even requires no Byzantine fault tolerance (BFT) protocol execution in happy paths, and the cost in unhappy paths is still not higher than a two-phase commit. Besides, we propose secure cross-shard certification methods. Handling b transactions, Kronos is proved to achieve cross-shard communication with low *cross-shard overhead* $\mathcal{O}(nb\lambda)$ (n for the shard size and λ for the security parameter). Notably, Kronos imposes no restrictions on BFT and does not rely on timing assumptions, offering optional constructions in various modules. Kronos could serve as a universal framework for enhancing the performance and scalability of existing BFT protocols. Kronos supports generic models, including asynchronous networks, and can increase the throughput by several orders of magnitude.

We implement Kronos using two prominent BFT protocols: asynchronous Speeding Dumbo (NDSS'22) and partially 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 [1]. 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 [2], privacy-preserving computation [3], and decentralized identity [4] to improve overall security and performance. Blockchain also drives the development of emerging domains such as Web3.0 [5] and Metaverse [6], requiring higher throughput and lower latency.

Scalability bottleneck and the potential of sharding. Practical applications reveal a significant challenge in traditional blockchains—*poor scalability* [7]. 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 [7] 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 [8]. Currently, significant attention has been directed towards blockchain sharding, with notable examples including [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20].

Sharding technology offers a promising solution for scalability enhancement, but it also comes with specific issues. 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 [9] (SP'18), RapidChain [10] (CCS'18), and Gearbox [21] (CCS'22) offering remarkable solutions.

Introduced a new scenario: cross-shard transactions. In particular, another critical issue is *cross-shard transaction processing*. Each shard separately manages a part of addresses according to specific assignment rules along with the *unspent*

Corresponding authors: Yuan Lu and Jianwei Liu.

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.230472
www.ndss-symposium.org

transaction outputs (UTXOs) or accounts associated with the shard, bringing in *cross-shard transactions* [9] 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 to ensure efficiency and consistency across multiple involved shards [22]. According to data from Ethereum ICOs[1], the value of multi-input transactions (including high-value *crowdfunding transactions* and *consolidated payments*) in 2024 has reached 1 billion USD in Ethereum (as of June). In a sharding blockchain, the proportion of cross-shard transactions involving 2 input shards and 1 output shard is estimated to exceed 99% as the network scales to 16 shards [10].

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*. Atomicity requires that transaction execution follows an *"all-or-nothing"* principle, i.e., either all involved shards commit it, or each operation is aborted.

As shown in Fig. 1, valid transaction tx_α requires *"all"*-execution, where utxo_1 and utxo_2 are both spent for tx_α. Conversely, if a request is incomplete or contains errors (e.g., missing/invalid signatures, or refers to a non-existent utxo_2 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). Ensuring the atomicity of cross-shard transaction processing is crucial, relying on rigorous consensus invocation in each shard and meticulous cross-shard cooperation with reliable message transfer.

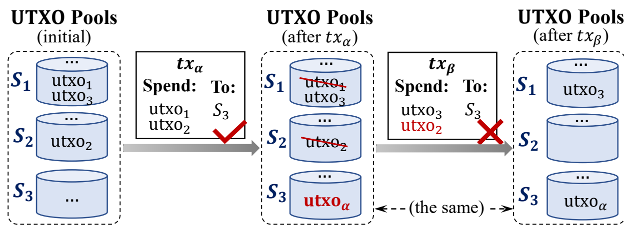


Fig. 1. Atomicity of cross-shard transaction processing.

Meanwhile, cross-shard transactions account for the vast majority of all transactions [10], so the *efficiency* of processing methods significantly influences the performance. Primary costs include two aspects: *intra-shard overhead* and *cross-shard overhead*. The former primarily focuses on the expenses of adopted BFT[2] and the BFT invocation times required to process a cross-shard transaction (batch). Cross-shard overhead revolves around the transfer of transactions and proofs between shards. A critical objective is to optimize both the intra-shard and cross-shard overheads for cross-shard transaction processing without compromising security and atomicity.

[1]https://coincodex.com/ico-calendar/ethereum/

[2]We use BFT to represent Byzantine fault tolerance protocol.

A. Remaining Issues of Prior Solutions

Q1: Weak atomicity and high overhead. For the atomicity property, existing schemes only achieve "weak atomicity". As shown in Table I, most schemes cannot tolerate malicious leaders and clients since they can refuse to correctly compute and transfer messages. Some schemes (Monoxide [18]) even cannot process multi-input cross-shard transactions.

As for overhead, Omniledger [9], Chainspace [14] (NDSS'18), ByShard [23], and Haechi [19] (NDSS'24, uses a method similar to Chainspace) 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 round, 2PC's twofold BFT increases overhead. RapidChain [10], Reticulum [20] (NDSS'24, uses RapidChain's method), and Monoxide [18] decrease the overhead of 2PC by spending funds directly to the payee. However, they cannot effectively process invalid multi-input transactions.

Some works try to process cross-shard transactions by introducing additional tools or special shard structures, leading to extra costs or new issues. *Reference shards* [11], [24] are adopted to transfer and commit cross-shard transactions through BFT, resulting in high extra cost. *Cross-shard nodes* joining multiple shards is adopted [15], [25], while ensuring that each cross-shard transaction corresponds to a cross-shard node for processing is challenging. *Merging shards* [26] 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. Danksharding [27] (proposed for Ethereum) utilizes one node as a block proposer. Multiple shards use *Data Availability Sampling* technique with *Reed-Solomon coding* and *KZG commitment* [28] to verify the block efficiently. Since only validators are divided into shards, Danksharding has no cross-shard transaction, supporting *Layer-2 Rollup* protocols.

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 [14], [29], [11], 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 [15], [18] 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.

Furthermore, existing solutions fail to ensure the reliability of critical cross-shard message transfer. In a Byzantine setting, certificates must be ensured to be sent by a source shard and reliably received by at least one honest party in a destination

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

System	Malicious Leader Tolerance	Malicious Client Tolerance	Atomicity*	IS-Overhead†	CS-Overhead (for b transactions)†	Genericity (of network settings)
Omniledger [9]	✗	✗		$2k\mathcal{B}$	$\mathcal{O}(b(\log b+\lambda))$	partially synchronous
Chainspace [14]	✓	✗		$2k\mathcal{B}$	$\mathcal{O}(n^2b\lambda)$	partially synchronous
ByShard [23]	✓	✗	✓✗	$2k\mathcal{B}$	$\mathcal{O}(n^2b\lambda)$	synchronous
RapidChain [10]	✗	✓	(cannot tolerant malicious	$k\mathcal{B}$	$\mathcal{O}(n^2b\lambda)$	synchronous
Sharper [29]	✗	✓	leaders and/or clients)	-♭	$\mathcal{O}(n^2b\lambda)$	partially synchronous
AHL [11]	✓	✗		$(2k+3)\mathcal{B}$	$\mathcal{O}(n^2b\lambda)$	partially synchronous
Pyramid [15]	✓	✗		$(k+1)\mathcal{B}$	$\mathcal{O}(n^2b\lambda)$	partially synchronous
Monoxide [18]	✗	✗	✓✗ (cannot process multi-input cross-shard transactions)	$k\mathcal{B}$ ($k=2$)§	$\mathcal{O}(nb\lambda)$	partially synchronous
Kronos-HTℵ **Kronos-VC**	✓	✓	✓	$k\mathcal{B}$◇	$\mathcal{O}(n\xi\lambda)$♮ $\mathcal{O}(nb\lambda)$	(partially) synchronous/ **asynchronous**

* "✓✗" represents "weak atomicity".

† "IS" and "CS" denote *intra-shard* and *cross-shard*. k is the number of shards involved in a cross-shard transaction. $\mathcal{B}$ denotes a BFT cost: $\mathcal{O}(n)$ in partially synchronous networks (e.g., Hotstuff [30]) and $\mathcal{O}(n^2)$ in asynchronous networks (e.g., FIN [31]).

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

§ Monoxide exclusively considers *single-input, single-output* cross-shard transactions, involving at most two shards, i.e., $k = 2$.

ℵ Both Kronos-HT and Kronos-VC denote our work, respectively adopting different cross-shard communication methods (refer to Section IV-E for details).

◇ $k\mathcal{B}$ is for valid transactions. Invalid transactions require no BFT in commonly occurring happy paths. IS-Overhead for rejecting an invalid transaction with x inputs in the unhappy path is $2(x-x')\mathcal{B}$, where x' denotes the number of unavailable input shards (refer to Section V-B for details).

♮ $\xi = \max(n\log m, n\log n, b)$ (m denotes the total number of shards). When $\max(n\log m, n\log n) < b$, the overhead of Kronos-HT and Kronos-VC are at the same level, both $\mathcal{O}(nb\lambda)$ (refer to Section V-B for details).

shard. However, existing solutions either delegate this task to clients or shard leaders [9], [15], [29], which can fail facing a malicious client/leader that refuses to send these messages, or resort to a rude "all-to-all" broadcast without further optimizations [14], [11], [10], resulting in high communication overhead $\mathcal{O}(n^2b\lambda)$ as shown in Table I.

Q3: Lack of research on a generic approach towards sharding blockchains. A *generic* sharding approach[3], supporting synchronous, partially synchronous, and asynchronous network settings, 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. For example, to ensure that cross-shard transactions are reliably received and collaboratively processed by all involved shards, trivially using timeouts does not work in asynchronous settings.

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 blockchain consensus that ensures security with atomicity and realizes optimized overhead.

A new sharding blockchain consensus pattern realizing security with optimal intra-shard overhead. We propose a *new pattern* 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, protecting against malicious clients. 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 funds are transferred to the payee through the buffer. For invalid transactions, in happy paths, comprehensive rejection is achieved through cross-shard multicasting. In unhappy paths, the spent inputs will be securely rolled back. Notably, Kronos is proved to achieve *optimal intra-shard overhead* $k\mathcal{B}$, 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 not higher than 2PC. Kronos satisfies security with *atomicity* under malicious clients and leaders.

Secure cross-shard certification methods realizing reliability with low cross-shard overhead. We propose a reliable cross-shard certification mechanism utilizing (1) *batch certification* and (2) *asynchronous reliable cross-shard transfer*. Batch certification proves multiple inputs in a single proof using Merkle tree or vector commitment [32]. The proof is constructed after a shard executes BFT to process requests. Reliable cross-shard transfer adopts $\mathcal{O}(n)$-to-$\mathcal{O}(n)$ multicasting with *erasure coding* to encode certified requests, and each node only needs to send one code block and proof. In particular, Kronos is proved to realize cross-shard message transmission tolerating malicious leaders and clients with low cross-shard overhead $\mathcal{O}(nb\lambda)$.

Generic sharding blockchain constructions supporting asynchronous networks. Kronos imposes no restrictions on the BFT employed in each shard and does not rely on any timing assumption. We provide an independent transaction verification interface invoked by intra-shard (partially) synchronous or asynchronous BFT, and batch certification methods for all network models. As for reliable cross-shard message transfer, Kronos provides communication-optimized $\mathcal{O}(n)$-to-$\mathcal{O}(n)$ for asynchronous networks and $\mathcal{O}(1)$-to-$\mathcal{O}(1)$

[3]Throughout the paper, we let "generic" refer to accommodation for any network environments, including synchronous, partially synchronous, and asynchronous networks.

for (partially) synchronous networks. Notably, Kronos can be a universal framework for enhancing the performance and scalability of existing BFT, supporting all kinds of network models, making it the first asynchronous sharding blockchain consensus. 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 [33], 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 partially synchronous BFT HotStuff [30] 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.

A. Challenges

Challenge 1: On processing pattern—Efficiently repairing weak atomicity of (multi-input) cross-shard transactions in even pure asynchrony. A primary issue in cross-shard transaction processing is overhead (left part of Fig. 2). In 2PC, input shards need to execute BFT two times. Removing one to reduce overhead violates atomicity since both phases involve status changes: from "available" to "locked", and from "locked" to "unlocked" or "spent", which need the execution of BFT.

Moreover, in crowdfunding transactions, inputs belong to multiple clients, including malicious ones. If submission is done by a single client, a malicious one can selectively send requests to a set of shards while neglecting others (*silence attack*). This results in some shards being unaware of the transaction. The same issue arises when submission is done by respective input-holding clients to their corresponding shards, leading to valid inputs being incorrectly spent or permanently locked (right part of Fig. 2). Employing *timeouts* or letting each client submit to relevant shards individually after obtaining each other's signatures is suitable for (partially) synchronous networks, but inapplicable to asynchronous ones.

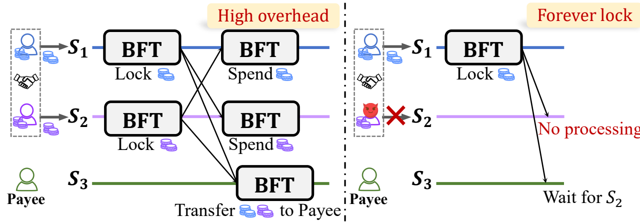


Fig. 2. Challenge 1: high overhead and weak atomicity. The left part shows 2PC, where each input shard requires two BFT rounds, leading to high intra-shard overhead. The right part illustrates that a request is not submitted to a shard by a malicious client, causing the shard to lock funds forever.

Challenge 2: On cross-shard certification—Realizing both *reliable* and *efficient* cross-shard communication in *asynchronous networks*. We identify a *disguise attack* by malicious leaders (left part of Fig. 3). 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 (proofs) 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 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 $\mathcal{O}(n)$-to-$\mathcal{O}(n)$ cross-shard broadcasting incurs significant cross-shard communication overhead $O(n^2b\lambda)$ (b for transaction number), which limits the scalability (right part of Fig. 3). 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.

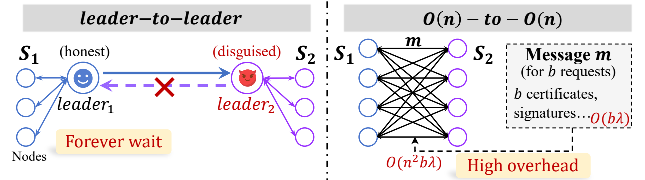


Fig. 3. Challenge 2: unreliable or verbose cross-shard certification. The left part depicts a leader disguise attack, where the leader acts honestly within a shard but refuses to forward messages across shards. The right part shows the high overhead of cross-shard communication.

B. Technical Contribution Overview

Our solutions to Challenge 1. We design a novel pattern for cross-shard transaction processing that achieves security and optimal intra-shard overhead. We observe that for input shards, 1 round of BFT instead of 2 rounds in 2PC is enough to spend the available inputs, but directly transferring funds to the payee is insecure. Therefore, we use a "buffer" as a transfer station to store and manage the spent inputs. To ensure the security, the buffer is jointly managed by the output shard nodes through $(f+1, n)$ threshold signatures [34] or multi-signatures [35]. Initially, input shards invoke BFT to deposit available inputs into the output shard buffer. For valid transactions, all inputs are available; the output shard executes BFT to transfer funds from the buffer to the payee. Consequently, only k times BFT are invoked (k denotes the number of shards involved in a transaction). For an invalid transaction, we notice that cross-shard multicasting is enough for transmitting unavailability proofs instead of BFT. At this point, if other shards have not processed it (*happy path*), they discard the request. Otherwise, if some 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.

To prevent silence attacks from malicious clients, our sharding blockchain consensus pattern employs a gathering-style request delivery method for cross-shard transaction requests. Each client submits a signed request to the output shard, which collects all relative clients' signed requests and transfers them to the involved shards. This ensures that all relevant shards receive and process the transaction.

Our solutions to Challenge 2. To design a secure cross-shard certification method with low overhead, 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.

To prevent disguise attacks and achieve asynchronous reliability, we adopt an improved $\mathcal{O}(n)$-*to*-$\mathcal{O}(n)$ cross-shard transfer method. To reduce communication overhead, we employ *erasure coding* to decrease the message size. The cross-shard requests are encoded to n code blocks, and each party is only responsible for sending one code block to parties in the output shard. To prevent Byzantine parties from sending fake code blocks, Kronos commits and proves the blocks as a *code tree* or *code vector*. Compared to a straightforward broadcast, the communication overhead of cross-shard batch certification decreases from $\mathcal{O}(n^2 b\lambda)$ to $\mathcal{O}(nb\lambda)$.

Our solutions vs. 2PC. Compared with 2PC, Kronos realizes full atomicity under asynchronous networks and tolerates malicious participants, including malicious clients and leaders. Besides, the buffer mechanism frees Kronos from the burden of locking, reducing the intra-shard overhead for valid transactions from $2k\mathcal{B}$ of 2PC to an optimal $k\mathcal{B}$. As for invalid transactions, the intra-shard overhead is not higher than 2PC.

III. Problem Formulation

In this section, we describe cryptographic primitives, notations, and our system models.

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 P_i has a key pair (sk_i, pk_i), a private function $\mathsf{ShareSig}$ and public functions $\mathsf{ShareVerify}$, $\mathsf{Combine}$ and Verify. sk_i is P_i's secret key for signing messages, and pk_i is P_i's public key share of a group public key gpk where gpk is used to verify signature combined from t parties' signature shares. Both the signature share and the combined threshold signature are *unforgeable* and the scheme is *robust* (a valid threshold signature on a message m is sure to be generated) [36]. See Appendix A for a formal definition.

Erasure coding. Let $0 \leq r \leq n$, a (r, n)-erasure coding scheme encodes a message M into n blocks using algorithm $\mathsf{ECEnc}(M, n, r) \rightarrow \{a_i\}_n$. The original message M can be reconstructed through decoding algorithm $\mathsf{ECDec}(T) \rightarrow M$, where T consists of at least r encoded blocks a_i. The encoded results $\{a_i\}_n$ by any party remain consistent, and M is certainly retrieved from r correct blocks [37].

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 $\mathsf{TreeCon}(D) \rightarrow (\mathsf{tree}, \mathsf{rt}, \mathsf{hp})$ in this paper, where D is the set of data blocks and 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 [38].

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* [39] [4] despite adversaries controlling the communication network and corrupting some parties. A secure BFT is defined as follows.

Definition 1 (Byzantine fault tolerance protocol)**.** *A Byzantine fault tolerance protocol (BFT) maintains a consistent ledger among a set of nodes. The protocol is secure if and only if it satisfies the following properties:*

- *Safety: If any two honest nodes respectively output committed proposals* p *and* p' *at the same position of the ledger, there must be* $p = p'$.
- *Liveness: If a proposal* p *is submitted to all honest nodes, then all honest nodes eventually commit* p.

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 (see Section IV-D for application in account model systems). 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 $\mathsf{type} = \textsc{sp}$) for input

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

shard spending, FINISH-TRANSACTION (denoted as FH-tx, within type = FH) for output shard committing, and BACK-TRANSACTION (denoted as BK-tx, within type = BK) for rolling back invalid execution.

Transaction waiting queue Q**.** Each node inside a 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 log $= tx_1 \parallel tx_2 \parallel \cdots$. Cross-shard transactions are recorded in output shards' ledgers.

C. System model

Network model. Each node 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 timing 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. 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. 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).

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. Sharding Blockchain Formulation

1) Sharding Blockchain: We design Kronos for a sharding blockchain system, where N nodes in the network are divided into m shards. Each shard S_i (where $i = 1, 2, \cdots, m$) includes nodes $\{P_j\}_{j\in[n]}$ (n is the shard size and $[n]$ denotes the integers $\{1, 2, \cdots, n\}$). Kronos operates in consecutive rounds. In each round, nodes within a shard run a BFT to process a batch of transactions where (some) input(s)/output(s) are managed by their shard. Committed transactions by each shard S_i are appended in an ordered manner to its shard ledger S_i.log where each transaction tx corresponds to a specific position h. Each valid intra-shard transaction committed by BFT is recorded in the corresponding shard ledger. Cross-shard transactions must be processed by all involved shards, and valid ones are only *recorded by the output shards*. Invalid transactions, involving invalid signatures or unavailable inputs, will not be recorded by any shard.

2) Security Goal: We define a secure sharding blockchain consensus by incorporating security properties outlined in [40]. Our definition extends the discussion to encompass invalid transaction request processing and introduces *atomicity*. The precise definition is as follows:

Definition 2 (Secure sharding blockchain consensus)**.** *A sharding blockchain consensus runs BFT in each shard in consecutive rounds to commit transactions, and each shard records the committed transactions in its shard ledger. The protocol is secure if and only if it satisfies the following properties:*

- *Persistence: If an honest party reports a transaction tx is at position h 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.*

To ensure each shard runs BFT correctly, *honest shard* must be satisfied, which means the Byzantine party number f in each shard cannot exceed the fault tolerance limit: $f/n < 1/3$ in partially synchronous or asynchronous networks, and $f/n < 1/2$ in synchronous networks. *Secure shard configuration*, which assigns nodes to different shards based on specific rules, is used in Kronos to realize honest shards. Configuration typically occurs at system initialization and at regular intervals after each shard operates for a period of time. Secure shard configuration [10], [14], [11], [21], [41] can be realized using public randomness and *Proof-of-Work/Proof-of-Stake*. Please refer to Appendix B for detailed shard configuration process.

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

Intra-shard communication overhead (IS-ω, Definition 3): 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 3 (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 4): CS-ω is determined by the adopted cross-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 4 (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. Initially, each shard initializes two threshold signature (or multi-signature) schemes 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) [42], [43], [44] (unnecessary when using multi-signature). Notably, the group public key of $(f+1, n)$-threshold signature scheme gpk^{t} 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. 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 of the cross-shard protocol. Fig. 4 gives the valid transaction processing pattern of Kronos. Clients initially submit their requests to corresponding output shards. Cross-shard requests with complete information are delivered to other involved shards (Step ❶). Upon receiving a cross-shard request, each input shard examines the input availability and spends available inputs to the output shard buffer through BFT (Step ❷). Processed cross-shard requests are certified to the corresponding output shards in batch using a $\mathcal{O}(n)$-to-$\mathcal{O}(n)$ *reliable cross-shard batch certification* (RCBC) method. After receiving and verifying the requests, honest nodes in the output shard store certified inputs in the shard buffer (Step ❸). Once the buffer has stored all inputs of a certain request, 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 transfer the inputs from buffer to the payee's address through BFT (Step ❹).

Fig. 5 shows invalid transaction processing pattern of Kronos. In case honest nodes in an input shard find an unavailable input, they do not process the request through BFT but sign to the invalid request and send the signatures to all other involved shards (Step ❷ #). If these signatures arrive at other input shards *before* they spend for the request, then these input shards can simply abort the processing, and output shards reject the request (the happy path of Step ❸ #). Otherwise, signatures are received *after* some input shards have spent for it, input shards return the inputs to their initial addresses through BFT, and output shards remove the stored inputs from the buffer to reject the request (the unhappy path of Step ❸ #).

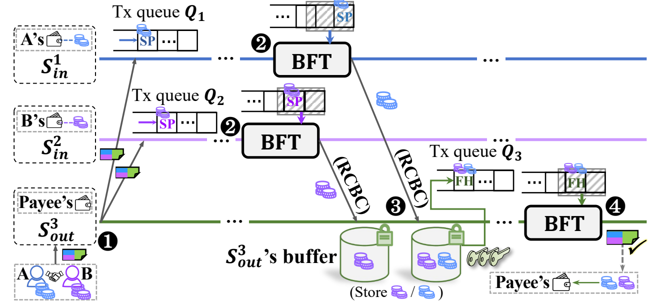


Fig. 4. Valid transaction processing pattern of Kronos.

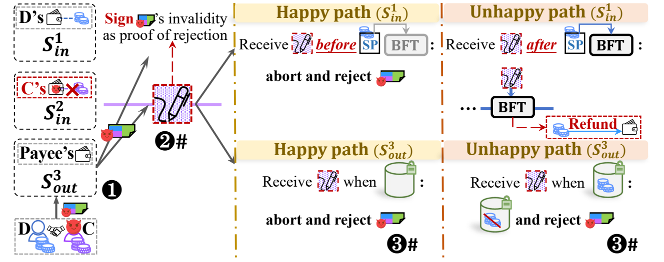


Fig. 5. Invalid transaction rejection pattern of Kronos.

C. Valid Transaction Processing of Kronos

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

Algorithm 1 shows the procedure of request delivery in Step ❶ . 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, TxCon$_{\mathsf{SP}}$ is invoked (Line 8).

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 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$_{\mathsf{SP}}$, Algorithm 2)

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

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 their unspent transaction output pool 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 5 (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 req$[id]$ is intra-shard, the output address pk is the payee's public key (Lines 11-12). In case 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_{SP})

```
    ► As a party P_i in shard S_c (Step ❷ ):
 1: upon receiving req[id] do
 2:   verify that:
 3:   • input I of req[id] managed by S_c holds I.utxo ∈ UTXO_c
 4:   • I.sig is valid                        ▷ verify the input availability.
 5:   if some I' verification fails and req[id] is intra-shard then
 6:     respond to the client that req[id] commitment failed
 7:   if some I' verification fails and req[id] is cross-shard then
 8:     execute request rejection message transmission (RRMT, Algorithm 5)
 9: upon verified req[id] do
10:   construct SP-tx[id] := (SP, id, I, O), where I is set of Is to be spent
11:   if req[id] is intra-shard then
12:     set O := (S_c, pk, v) where pk is the public key of payee  ▷ spend
      to the payee directly.
13:   otherwise, O := (S_out, gpk^t_out, I.v) where gpk^t_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 TxVerify(tx)
16:   if tx.type = SP then
17:     if for each I_i ∈ I where I_i = ⟨S_c, utxo_i, sig_i⟩, utxo_i ∈ UTXO
      and sig_i is verified with utxo_i.pk, return 1
18:     otherwise, return 0 and execute RRMT (Algorithm 5)
```

Algorithm 3 shows how each shard processes transactions in Q. To ensure that transactions are executed in order of submission, BFT processes transactions selected from Q in order in consecutive round ℓ. 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 req$[id]$ is intra-shard, honest parties record SP-tx$[id]$ in the current shard ledger S_{in}.log, update the output to $\mathsf{UTXO}_{\mathsf{in}}$, and finalize the request processing by responding to the client (Lines 6-10). Otherwise, id is put into cID_ℓ and cross-shard req$[id]$ 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.

Step ❸ : Reliable 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 prove 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*-$\mathcal{O}(n)$, where each party multicasts processed requests in batch to all parties in the output shard, ensuring reliability

Algorithm 3 Intra-shard workflow and state update

```
  Let BFT_ℓ execute in consecutive round number ℓ.
  Initial processed cross-shard request ID set cID_ℓ = ∅ before BFT_ℓ.
    ► As a party P_i in shard S_c
 1: for Q ≠ ∅ do
 2:   take out at most b transactions {tx}_b from the top of Q  ▷ in order
 3:   execute BFT_ℓ({tx}_b) with function TxVerify(tx)      ▷ SP, FH, BK
    transactions verification is described in Algorithm 2, 4, 7, respectively
 4:   output transaction set TXs_ℓ committed by BFT_ℓ({tx}_b)
 5:   set ℓ ← ℓ + 1
 6: upon receiving committed TXs_ℓ do                        ▷ state update
 7:   for each tx_i ∈ TXs_ℓ do
 8:     if tx_i.type = SP then                  ▷ Step ❷ and Algorithm 2
 9:       update UTXO ← UTXO\I[tx_i].utxo if I[tx_i].utxo ∈ UTXO
10:       update log ← log || tx_i and UTXO ← UTXO ∪ tx_i.O if tx_i is
    for an intra-shard request
11:       set cID_ℓ ← cID_ℓ ∪ tx_i.id if tx_i is for a cross-shard request
12:     if tx_i.type = FH then                  ▷ Step ❹ and Algorithm 4
13:       update log ← log || tx_i and UTXO ← UTXO ∪ tx_i.O
14:     if tx_i.type = BK then                  ▷ Step ❸ # and Algorithm 7
15:       update UTXO ← UTXO ∪ tx_i.O
16:   if cID_ℓ ≠ ∅ then
17:     execute reliable cross-shard batch certification (Algorithm 8)
```

without any extra assumption. To reduce the overhead, we adopt *erasure coding* to encode the requests, and each party only sends a single code block committed by shard parties. Once output shard parties receive enough messages from an input shard, they verify the commitment, decode processed requests, and store inputs of committed requests into their shard buffer. Please refer to Section IV-E for the specific batch certification operations.

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

In Algorithm 4, when an honest party P_i in S_{out} receives all inputs of request req$[id]$ (confirming its validity), the party signs to req$[id]$ inputs using sk^t_i 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 req$[id]$, indicating at least one honest party has received integral inputs, req$[id]$'s inputs in $\mathsf{buffer}_{\mathsf{out}}$ become accessible and capable of being transferred by a FINISH-TRANSACTION, FH-tx$[id]$. The input of FH-tx$[id]$ comprises all inputs of req$[id]$ stored in $\mathsf{buffer}_{\mathsf{out}}$ along with the $(f+1, n)$-threshold signature (Lines 4-6). Then FH-tx$[id]$ is ready to be processed by BFT in Algorithm 3 (Line 7). The verification of FH-tx$[id]$ is done by function TxVerify (Lines 8-11). When a certain round BFT outputs FH-tx$[id]$, it is recorded in the output shard ledger and added to $\mathsf{UTXO}_{\mathsf{out}}$. Then the stored inputs of req$[id]$ are removed from buffer, and S_{out} responds 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 *content-incorrect*, where the utxo is non-existent, or sig is invalid. This incorrectness can only be verified by the input shard

Algorithm 4 FINISH-TRANSACTION construction (TxCon_{FH})

▶ As a party P_i in shard S_c: (Step ❹)
1: **for each** req$[id]$ where every I has been stored in buffer **do** ▷ req$[id]$ is valid and ready to be committed
2: **sign** $s_i{}^{\text{FH}} := \text{ShareSig}(sk_i^{\text{t}}, \text{H}(\{I\}))$ and multicast $(id, s_i{}^{\text{FH}})$ among S_c. ▷ Apply to execute 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 $\text{ShareVerify}(\text{H}(\{I\}), \{j, s_j\}) = 1$ **do**
5: **compute** $\sigma^{\text{FH}}[id] := \text{Combine}(\text{H}(\{I\}), \{j, s_j\}_{f+1})$ ▷ serve as the validity proof of FH-tx$[id]$
6: **construct** FH-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** FH-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 σ^{FH}
10: **if** $\text{Verify}(\text{H}(\{I_i\}), \sigma^{\text{FH}}) = 1$ where $\mathbf{I} = (\{I_i\}, \sigma^{\text{FH}})$, return 1
11: **otherwise**, return 0

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 5, upon receiving req$[id']$, each honest party in S_c identifies that I' is unavailable and signs a REJECT-MESSAGE m_{RJ} for 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 5 Request rejection message transmission (RRMT)

Let I' denotes the unavailable input of request req$[id']$ managed by S_c

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

Step ❸ #: Invalid request rejection (by input and output shards, in Case 1 or Case 2).

After receiving the reject messages, each shard either enters Case 1 (the happy path) or Case 2 (the unhappy path) according to whether an input shard has processed the transaction.

Case 1 (the happy path): Abort processing.

Output shard operations (Algorithm 6, Lines 1-5, 7): Upon receiving $n-f$ reject message of req$[id']$, honest parties in output shard compute threshold signature to verify the message (Lines 1-2). If there are no corresponding inputs of req$[id']$ in the buffer, abort the invalid request (Lines 4-5). Afterwards, S_{out} responds to clients that req$[id']$ is rejected (Line 7).

Input shard operations (Algorithm 6, Lines 8-12): Upon receiving $n-f$ reject message for invalid req$[id']$, honest parties in S_{in} verify the threshold signature (Lines 8-10). Then, each party checks whether req$[id']$ has already been processed. If not, honest parties quit executing it and eliminate SP-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 (where the round of communication is at least one in synchronous networks and two or three in partially synchronous or asynchronous networks). Therefore, the happy path often occurs with no BFT being "wasted on" invalid requests.

Algorithm 6 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 $n-f$ $m_{\text{RJ}}[id']$ from distinct parties P_j in an input shard of req$[id']$ **do**
2: **compute** $\sigma^{\text{RJ}}[id'] := \text{Combine}(m_{\text{RJ}}[id'], \{(j, s_j)\}_{n-f})$
3: **if** $\text{Verify}(m_{\text{RJ}}[id'], \sigma^{\text{RJ}}) = 1$ **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$\setminus\{id', \mathbf{I}\}$ ▷ 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 $n-f$ $m_{\text{RJ}}[id']$ from distinct parties P_j in shard S_{in} **do**
9: **compute** $\sigma^{\text{RJ}}[id'] := \text{Combine}(m_{\text{RJ}}[id'], \{(j, s_j)\}_{n-f})$
10: **if** $\text{Verify}(m_{\text{RJ}}[id'], \sigma^{\text{RJ}}) = 1$ **then**
11: **if** SP-tx$[id'] \in \text{Q}_{\text{in}}$ **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 7

Algorithm 7 BACK-TRANSACTION construction (TxCon_{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 $= (\text{BK}, id, \mathbf{I}, \mathbf{O})$, where $\mathbf{I}$ = SP-tx$[id]$.$\mathbf{O}$ with T-SIG $= \sigma^{\text{RJ}}$. $\mathbf{O} = (S_c, pk, v)$ 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 σ^{RJ}
6: **if** $\text{Verify}(m_{\text{RJ}}, \sigma^{\text{RJ}}) = 1$ where $\mathbf{I}$.T-SIG $= \sigma^{\text{RJ}}$, return 1
7: **otherwise**, return 0

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

Output shard operations (Algorithm 6, Lines 1-3, 6-7): If there is input in the buffer sent from the input shard, output shard's parties remove the input of invalid req$[id']$ from $\text{buffer}_{\text{out}}$, and respond to the client that req$[id']$ is rejected (Lines 6-7).

Input shard operations (Algorithm 7): As Fig. 5 shows, S_2 suffer a terrible latency or adversary delay, and S_1 has spent for req$[id']$ with SP-tx$[id']$. For a comprehensive rejection, S_1 returns the spent input to the initial payer through a BACK-TRANSACTION BK-tx$[id']$ with the received threshold signature as T-SIG, where the utxo is the output of the committed SP-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-tx$[id']$ is output by BFT in S_1, every honest party update UTXO with BK-tx$[id']$.$\mathbf{O}$, ensuring a comprehensive rollback (Algorithm 3, Lines 14-15).

Regarding account model systems. The cross-shard processing pattern of Kronos is applicable not only to the UTXO model but also to the account model. The account model allows for changes to the state, which typically refers to the balances,

data, and storage (memory). In account model systems, the input shard runs BFT, updates the account state, and sends update proof to the involved shard. For valid transactions, the output shard's buffer receives all account state updates and completes the final state update (e.g., transferring funds to the payee). For invalid transactions, the output shard abandons the operation, while the input shard, which has already completed the state update, rolls back the state by running BFT.

E. Optimizing Reliable Cross-Shard Batch Certification

In Step ❸ , to prove spent inputs to output shards efficiently, Kronos adopts *hybrid-tree-based reliable cross-shard batch certification* (HT-RCBC) using erasure coding and Merkle tree. The certification process is shown in Fig. 6. 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.

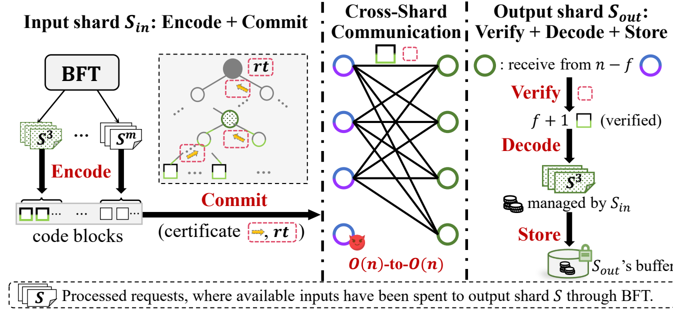


Fig. 6. Reliable cross-shard batch certification.

Input shard operations (Algorithm 8, 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 $\mathcal{O}(b\lambda/n)$) 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 $m^i_{\text{BF}} = (\langle \text{rt}, s^{\text{BF}}_i \rangle, a_i, \text{l-hp}_i, \text{hp}_{\text{out}})$ to parties in S_{out}, where s^{BF}_i is the signature share of rt using P_i private key sk^{T}_i, and l-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 $\mathcal{O}(n\xi\lambda)$ for b transactions where $\xi = \max(n\log m, n\log n, b)$.

Output shard operations (Algorithm 8, Lines 8-18): Lines 8-18 show how honest parties in output shard S_{out} learn and

Algorithm 8 Hybrid-tree-based reliable cross-shard batch certification (HT-RCBC)

Let cID_c link IDs where corresponding SPEND-TRANSACTION are committed to be spent to S_c by the latest round BFT

▶ As a party P_i in input shard S_{in} (Step ❸):
1: **for each** $(c \in [m]) \wedge (\text{cID}_c \neq \bot)$ **do** ▷ requests from S_c are processed
2: **encode** $\{a^c_i\}_n \leftarrow \text{ECEnc}(\text{cID}_c, n, n-2f)$
3: **compute** $(\text{l-tree}_c, \text{l-rt}_c, \{\text{l-hp}_c\}) \leftarrow \text{TreeCon}(\{a^c_i\}_n)$ ▷ lower code tree for coding examination
4: **compute** $(\text{tree}, \text{rt}, \{\text{hp}\}) \leftarrow \text{TreeCon}(\{\text{l-rt}_c\}_{c\in[m]})$ ▷ hybrid higher tree for batch certification, with a maximum of m leaves l-rt_c
5: **sign** $s^{\text{BF}}_i \leftarrow \text{ShareSig}(sk^{\mathsf{T}}_i, \text{rt})$
6: **for each** shard S_c with a leaf l-rt_c **do**
7: **send** $m^i_{\text{BF}} = (\langle \text{rt}, s^{\text{BF}}_i \rangle, a_i, \text{l-hp}_i, \text{hp}_c)$ to parties in S_c.

▶ As a party P_i in output shard S_{out} (Step ❸):
8: **upon** receiving m^j_{BF} from P_j of shard S_{in} **do**
9: **parse** $m^j_{\text{BF}} = (\langle \text{rt}, s^{\text{BF}}_j \rangle, a_j, \text{l-hp}_j, \text{hp}_{\text{out}})$
10: **check** that $(\text{l-hp}_j, \text{hp}_{\text{out}})$ is a valid hash path from leaf a_j to root rt, otherwise discard
11: **upon** receiving $n-f$ valid m_{BF} from distinct parties P_k of shard S_{in} **do**
12: **compute** $\sigma \leftarrow \text{Combine}(\text{rt}, \{(k, s^{\text{BF}}_k)\}_{n-f})$
13: **if** $\text{Verify}(\text{rt}, \sigma) = 1$ **then**
14: **interpolate** a'_j from any $n-2f$ m_{BF} received
15: **recompute** Merkle root rt' and if $\text{rt}' \neq \text{rt}$ then abort
16: **decode** $\text{ID}_{\text{out}} \leftarrow \text{ECDec}(\varphi_{\text{out}}, \{a_k\}_{n-2f})$
17: **update** $\text{buffer} \leftarrow \text{buffer} \bigcup \{(id, I)\}$ where each $id \in \text{ID}_{\text{out}}$ and I is managed by S_{in}
18: **otherwise** wait for other valid m_{BF} ▷ there must be $n-f$ m_{BF} that can pass verification under honest shard configuration

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 $(\text{l-hp}_j, \text{hp}_{\text{out}})$ (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). To ensure code block correctness, 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 $\text{buffer}_{\text{out}}$ and awaits integral inputs (Line 17).

Alternative cross-shard batch certification methods. When using Merkle tree, the proof size for a code block is $\mathcal{O}(\log n + \log m)$ (see the detailed complexity analysis of HT-RCBC in Appendix E). We propose an alternative method, *vector-commitment-based reliable cross-shard batch certification* (VC-RCBC), in Appendix C, which uses compact vector commitments to commit the code blocks, reducing the commitment and proof size to constants. HT-RCBC is based on collision-resistant hash functions. VC-RCBC relies on a trusted setup, and needs assumptions of the commitment schemes used (e.g., *computational Diffie-Hellman assumption* when adopting [32]). For (partial) synchronous networks, it's applicable that only shard leaders send and receive cross-shard messages utilizing cross-shard view-change.

Batch proof approaches in different networks. Both HT-RCBC and VC-RCBC require shard parties to sign the com-

mitment (rt of HT-RCBC or vector commitment C of VC-RCBC) 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 (partially) synchronous environments, where the adopted BFT (e.g., [30], [45], [46]) 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-BFT*). However, in asynchronous networks, the signing is necessary. By the FLP "impossibility" [47], asynchronous BFT (e.g., [48], [49], [50], [51], [52], [33]) must run *randomized* subroutines to ensure security, leading to uncertainty about the committed transactions. Hence, the tree (resp. vector) can only be constructed and signed *after* BFT (*batch-proof-after-BFT*).

V. Security and Complexity Analysis

A. Security Analysis

We prove that Kronos satisfies the security properties of persistence, consistency, atomicity, and liveness indicated in Definition 2. Due to page limitations, we provide theorems that Kronos satisfies each property and its associated guarantees. Please see Appendix D for detailed proofs.

Theorem 1 (Persistence)**.** *If an honest party P_i in shard S_c reports a transaction tx is at position h of P_i's shard ledger $S_c.\mathsf{log}_i$ in a certain round, then whenever tx is reported by any honest party P_j in shard S_c, it will be at the same position h in $S_c.\mathsf{log}_j$.*

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-Transaction. ■

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 batch certification. ■

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 E 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 k times, realizing the lower bound of intra-shard communication overhead $IS\text{-}\omega = k\mathcal{B}$, which is optimal for a secure sharding blockchain as per Definition 2.*

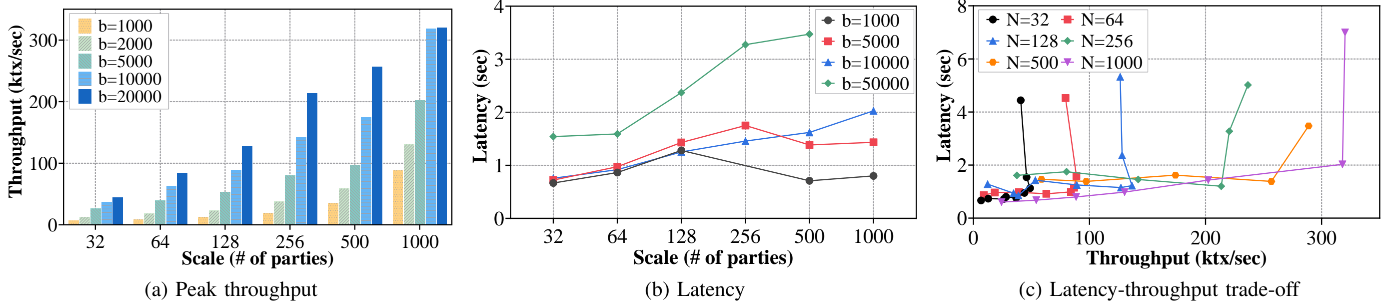
Overhead of happy and unhappy paths. The overhead declared in Theorem 5 pertains to *valid transaction processing*. For invalid transactions, intra-shard overhead of the happy path is $IS\text{-}\omega = 0$, as invalid transactions only need multicasting instead of BFT. In the unhappy path of rejecting an invalid transaction with x input shards, we assume that x' among them manage unavailable inputs and other $x - x'$ input shards have spent on it before receiving a Reject-message (e.g., $x = 3$ and $x' = 2$). Since only input shards with available inputs run BFT to spend and refund funds, the intra-shard overhead is $IS\text{-}\omega = 2(x - x')\mathcal{B}$. In 2PC, input unavailability is proved by BFT. The available inputs are locked and unlocked by 2 BFT rounds, leading to $IS\text{-}\omega = (2x - x')\mathcal{B}$. Kronos's unhappy path needs BFT x' rounds fewer than 2PC. Moreover, the happy path occurs *frequently* due to the fast one-round multicast-based rejection transmission, while the unhappy path is rare. Please refer to Appendix E for detailed complexity analysis.

Cross-shard communication overhead. Definition 5 describes the reliability property for cross-shard transmission. Theorem 6 depicts that Kronos realizes a low overhead of $CS\text{-}\omega = \mathcal{O}(nb\lambda)$ where n denotes the shard size, b denotes transaction batch size and λ is the security parameter.

Definition 5 (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 $CS\text{-}\omega = \mathcal{O}(nb\lambda)$, and realizes cross-shard message transmission reliability as per Definition 5.*

Comparison of HT-RCBC and VC-RCBC. In Kronos, the cross-shard communication overhead $CS\text{-}\omega$ when adopting HT-RCBC is $\mathcal{O}(n(b + n\log m + n\log n)\lambda)$, and $\mathcal{O}(nb\lambda)$ when using VC-RCBC. VC-RCBC can be utilized for lower communication cost if the relationship among batch size b, shard size n, and shard number m holds that $b < \min(n\log m, n\log n)$.



(a) Peak throughput

(b) Latency

(c) Latency-throughput trade-off

Fig. 7. Performance of sKronos in the WAN setting.

Otherwise, the overhead is at the same level. (Refer to Appendix E for more details.)

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

Implementation details. We program the implementations of Kronos and 2PC in the same language Python. All libraries and security parameters required in cryptographic implementations are the same. All nodes are assigned into shards, each adopting Speeding Dumbo [33] (an efficient and robust asynchronous BFT) 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 partially synchronous protocol, HotStuff [30]. All hash functions are instantiated using SHA256 [53]. 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.

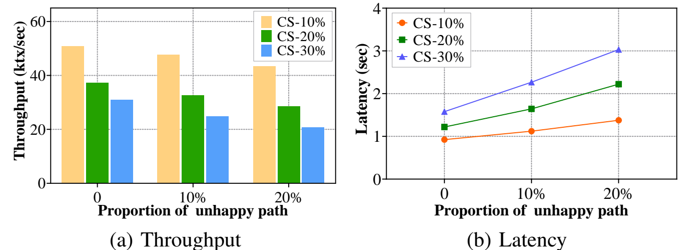
Setup on Amazon EC2. We implement sKronos, s2PC, and hKronos among Amazon EC2 c5.4xlarge instances, which are equipped with 16 vCPUs and 32GB main memory. The performances are evaluated with varying scales at $N = 32$, 64, 128, 256, 500, and 1000 nodes. The proportion of cross-shard transactions varies from 10% to 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.

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 $b = 1$k to 50k for evaluation (4 nodes in each shard). This reflects how well Kronos performs in realistic scenarios.

As illustrated in Fig. 7a, sKronos demonstrates scalability, showcasing an increasing throughput as network scales and achieving a peak throughput of 320.2 ktx/sec with $N = 1000$, $b = 20$k. As the network scales, the optimal batch size for achieving peak throughput decreases. Fig. 7b 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 $b \leq 10$k, highlighting the effectiveness of Kronos for latency-critical applications, even at a large scale.

Throughput-latency trade-off. Fig. 7c 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.



(a) Throughput

(b) Latency

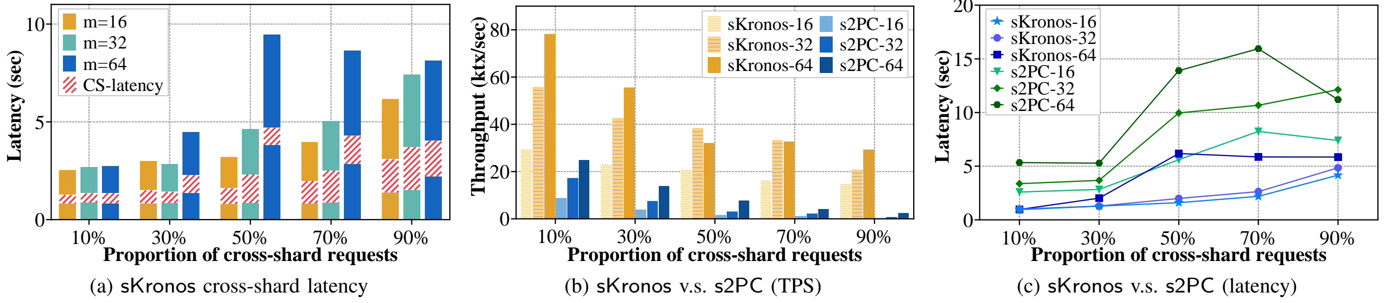
Fig. 8. Evaluation of unhappy paths.

Evaluation of unhappy paths. Fig. 8 illustrates the impact of unhappy paths on performance, where 0-20% cross-shard requests are invalid and rejected in unhappy paths ($N = 32$). Cross-shard request proportions are set as 10%, 20% and 30% (denoted as "CS-10%/20%/30%"). As shown in Fig. 8a, an increase in unhappy path proportion leads to a slight decrease in overall throughput, since some input shards run 2 BFT rounds to spend and refund funds for invalid transactions which are not counted in throughput. Fig. 8b shows that while latency slightly increases as the unhappy path proportion grows, it remains at a relatively low level.

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

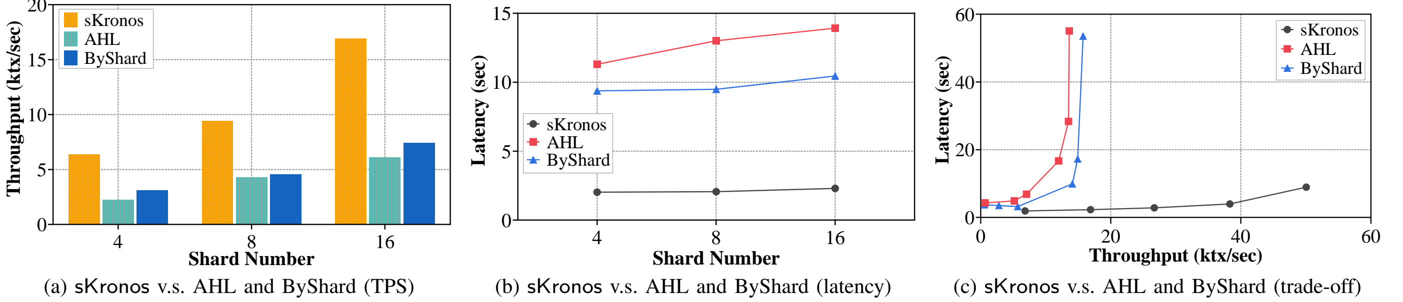
To analyze how well Kronos handles cross-shard requests, we evaluate the time cost of each processing step in sKronos. Besides, we compare throughput and latency of sKronos and s2PC with varying cross-shard transaction proportions.

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. 9a. Once delivered to every involved shard, a valid cross-shard request undergoes three distinct steps: input shard spending, cross-shard certification, and output shard buffer committing.



(a) sKronos cross-shard latency (b) sKronos v.s. s2PC (TPS) (c) sKronos v.s. s2PC (latency)

Fig. 9. 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. 10. Comparison with other solutions.

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.

The experimental results reveal the cost of cross-shard cooperation highlighted in red (with the same batch size b = 10k). 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. 9b and Fig. 9c 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× that of s2PC when 10% of requests are cross-shard, approximately 4× when the proportion is 50%, and an impressive 12× when the proportion is 90%! For latency, sKronos incurs at most half the time cost of s2PC (in all cross-shard proportion cases).

Comparison with other sharding blockchains. We also compare sKronos with state-of-the-art systems, AHL [11] and ByShard [23]. Each shard comprises 30 nodes. Fig. 10 demonstrates that the throughput of sKronos exceeds ByShard's by 2.3× and AHL's by 2.7×, 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. 11 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× higher peak throughput than s2PC while maintaining latency around 1 sec. hKronos throughput is much higher than s2PC by an order. These results indicate that Kronos is generic in any network environment with various BFT for enhancing blockchain scalability.

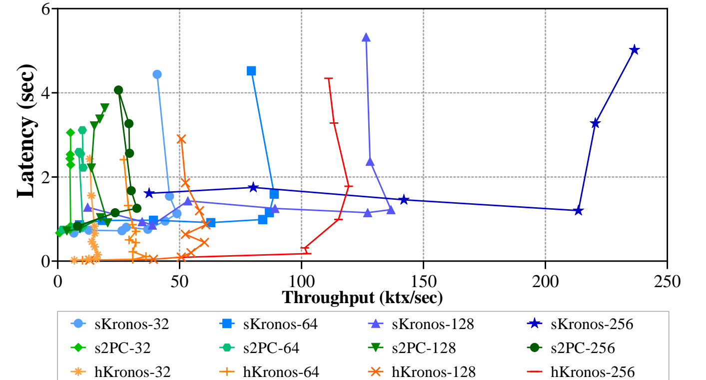


Fig. 11. Latency-throughput trade-off of several consensuses.

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 blockchain consensus pattern realizes atomicity with optimal intra-shard overhead. The concrete cross-shard batch certification construction realizes reliability with low cross-shard overhead. Implementation results demonstrate Kronos's outstanding scalability, surpassing existing solutions and 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, 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.

Acknowledgment

The authors would like to thank the reviewers for the valuable comments. This work was supported by the National Key R&D Program of China (2021YFB2700200), the National Natural Science Foundation of China (U21B2021, U22B2008, U2241213, 62202027, 61972018, 61932014, 62472015), the Young Elite Scientists Sponsorship Program by China Association for Science and Technology (2022QNRC001), and the Beijing Natural Science Foundation (M23016).

References

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

[2] 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.

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

[4] 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.

[5] 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.

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

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

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

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

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

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

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

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

[27] V. Buterin, "Proto-danksharding faq," 2022, https://notes.ethereum.org/@vbuterin/proto_danksharding_faq.

[28] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *ASIACRYPT'10*. Springer, 2010, pp. 177–194.

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

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

[31] 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.

[32] D. Catalano and D. Fiore, "Vector commitments and their applications," in *PKC'13*. Springer, 2013, pp. 55–72.

[33] 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.

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

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

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

[37] 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.

[38] 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.

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

[40] 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.

[41] 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.

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

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

[44] 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.

[45] 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.

[46] 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.

[47] 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.

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

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

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] S. Das, V. Krishnan, I. M. Isaac, and L. Ren, "Spurt: Scalable distributed randomness beacon with transparent setup," in *SP'22*. IEEE, 2022, pp. 2502–2517.

[55] Y. Liu, J. Liu, Y. Hei, W. Tan, and Q. Wu, "A secure shard reconfiguration protocol for sharding blockchains without a randomness," in *TrustCom'20*. IEEE, 2020, pp. 1012–1019.

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 involving 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 parameters λ, n, t, the algorithm 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 signer 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.

APPENDIX B
SECURE SHARD CONFIGURATION

In the following, we provide a common method to realize secure shard configuration. At system initialization, each node requesting to join the sharding blockchain computes a solution to a *Proof-of-Work* (PoW) problem using a *public randomness* η as a puzzle, and sends the solution to a reference committee for identity registration. After verifying these solutions, the reference committee executes a *Pseudorandom Permutation* function (PRP), where $\eta_i \leftarrow \mathsf{PRP}(\eta, i), i = 1, 2, \cdots N$. N is the total number of identified nodes. PRP generates a set of random values $\{\eta_i\}_{i \in [N]}$ (using η as a seed) for each identified node P_i using respective identifier i. The reference committee then assigns nodes to shards based on their random values through consensus. The assignment is determined by some specified rules, such as assigning node P_i with random value η_i to shard c where $\eta_i \bmod n \equiv c$. n is the shard size. Nodes allocated to the same shard run a DKG to obtain their keys and begin to process transactions using BFT.

During system operation, shards must be reconfigured at regular intervals to prevent any shards from being controlled by adversaries. The reconfiguration process is similar to the initial configuration. Before each reconfiguration, nodes jointly run a *public randomness generation* protocol [54] to generate new randomness η'. When the reconfiguration starts, nodes compute PoW solutions based on η' and send them to the reference committee, which runs BFT and determines the new node allocation. Each node that undergoes a shard change requests historical states from previous members of its new shard before processing transactions. Besides, the reference committee can also be performed by any shard committee [55].

APPENDIX C
VECTOR-COMMITMENT-BASED RELIABLE CROSS-SHARD BATCH CERTIFICATION

The method given in Section IV-E 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 and shard size n. We propose another method that utilizes *vector commitment* technology, maintaining a concise message size regardless of either batch size or shard number.

Algorithm 9 Vector-commitment-based reliable cross-shard batch certification (VC-RCBC)

Let 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_{in} (Step ❸):
1: **for each** $(c \in [m]) \wedge (\mathsf{cID}_c \neq \perp)$ **do**
2: **encode** $\{a_i^c\}_{[n]} \leftarrow \mathsf{ECEnc}(ID_c, n, n-2f)$
3: **construct** vector $\boldsymbol{vec} = \left((a_1^1, \cdots, a_n^1), \cdots, (a_1^m, \cdots, a_n^m)\right)$
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_{out} (Step ❸):
9: **upon** receiving m_{BF}^j from P_j of shard S_{in} **do**
10: **parse** $m_{\text{BF}}^j = (\langle C, s_j^{\text{BF}} \rangle, a_j, \Lambda_j)$
11: **check** that $\mathsf{ComVrf}(\Lambda_j, C, a_j) = 1$, otherwise discard.
12: **upon** receiving $n-f$ valid m_{BF} from distinct parties of shard S_{in} **do**
13: **compute** $\sigma \leftarrow \mathsf{Combine}(C, (k, s_k^{\text{BF}}))$
14: **if** $\mathsf{Verify}(C, \sigma) = 1$ **then**
15: **decode** $\mathsf{ID}_{\mathsf{out}} \leftarrow \mathsf{ECDec}(\varphi_{\mathsf{out}}, \{a_k^{\mathsf{out}}\}_{n-2f})$
16: **update** buffer $\leftarrow$ buffer $\bigcup\{(id, I)\}$ where each $id \in \mathsf{ID}_{\mathsf{out}}$ and I is managed by S_{in}
17: **else** wait for other valid m_{BF}

Input shard operations (Algorithm 9, Lines 1-8): Similar

to Method 1, parties divide processed requests and encode $\{a_i^c\}_n$ for each output shard S_c (Lines 1-2). Overall code blocks constitute a $\mathcal{O}(mn)$-sized vector $\boldsymbol{vec} = \big((a_1^1, \cdots, a_n^1), \cdots, (a_1^m, \cdots, a_n^m)\big)$, which is then committed through VecCom with a succinct-length commitment C (Lines 3-4). Each a_i^c has a succinct-length proof Λ_i^c, which validates that a_i^c is an element of the vector committed by C. After signing to C with sk_i^{T}, an honest party P_i in S_{in} sends $m_{\text{BF}}^c = (\langle C, s_i^{\text{BF}}\rangle, a_i^c, \Lambda_i^c)$ to parties in S_{out} with a succinct message size of $\mathcal{O}(b\lambda/n)$ (Lines 5-8).

Output shard operations (Algorithm 9, 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 $m_{\text{BF}}^j = ((\langle C, s_j^{\text{BF}}\rangle, a_j, \Lambda_j))$ 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 $\mathsf{buffer}_{\mathsf{out}}$ for future transfer or return (Line 16).

Appendix D
Security Analysis

Theorem 1 (Persistence)**.** *If an honest party P_i in shard S_c reports a transaction tx is at position h of P_i's shard ledger $S_c.\mathsf{log}_i$ in a certain round, then whenever tx is reported by any honest party P_j in shard S_c, it will be at the same position h in $S_c.\mathsf{log}_j$.*

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 using method given in Appendix B, each shard is configured to be honest shard, with the number of Byzantine parties f holds that $3f+1 \leq n$ in partially synchronous and asynchronous networks where n is the shard size. In synchronous networks, the limitation is relaxed to $2f+1 \leq n$. The secure shard configuration ensures BFT operates with both safety and liveness successfully.

In a given shard S_c, a secure BFT with an external function for transaction verification is employed. If honest party P_i outputs transaction tx in a committed set $\mathsf{TXs}_r[i]$ in a BFT round r, then for any other honest party P_j in S_c outputting $\mathsf{TXs}_r[j]$ in the same BFT round, it must hold that:

$$(tx \in \mathsf{TXs}_r[j]) \wedge (\mathsf{TXs}_r[j] = \mathsf{TXs}_r[i])$$

Upon receiving committed transactions $\mathsf{TXs}_r[j]$ from BFT_r, honest P_j in S_c operates $S_c.\mathsf{log}_j \leftarrow S_c.\mathsf{log}_j \parallel \mathsf{TXs}_r[j] (tx \in \mathsf{TXs}_r[j])$ to append committed $\mathsf{TXs}_r[j]$ to the shard ledger log_c. 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

$$tx_1.\mathbf{I} \cap tx_2.\mathbf{I} = \overline{I}, \overline{I}.\mathsf{utxo} = \overline{\mathsf{utxo}}$$

$\overline{\mathsf{utxo}}$ is the conflicting unspent transaction output spent by both tx_1 and tx_2. 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 inside a shard check whether there are conflicting transactions when receiving committed transactions output by each BFT before recording, tx_1 and tx_2 cannot be committed in the same round BFT by a shard. Therefore, the conflicting tx_1 and tx_2 can only be committed in different BFT rounds or by different shards:

$$(tx_1.\mathsf{type} = tx_2.\mathsf{type} = \textsc{sp}) \wedge (tx_1.\mathbf{I} \cap tx_2.\mathbf{I} \neq \varnothing) \iff$$
$$\exists\ \overline{\mathsf{utxo}} \text{ s.t., } \overline{\mathsf{utxo}} \in S_c.\mathsf{UTXO}_u \cap S_d.\mathsf{UTXO}_v\ (c \neq d \vee u \neq v)$$

$S_c.\mathsf{UTXO}_u$ and $S_d.\mathsf{UTXO}_v$ are the unspent transaction output pool states of shards S_c and S_d respectively running the uth round BFT_u and vth round BFT_v. For each shard S_c, while $S_c.\mathsf{UTXO}$ 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 $S_c.\mathsf{UTXO}$ and no one can spend it again. The states of $S_c.\mathsf{UTXO}$ must satisfy that:

$$\forall u \neq v,\ S_c.\mathsf{UTXO}_u \cap S_c.\mathsf{UTXO}_v = \varnothing$$

tx_1 and tx_2 cannot be committed in different BFT rounds by any honest shard S_c. Therefore, SP-tx conflicting can never occur inside any shard. Because one intra-shard request cannot have two different output shards, which holds that:

$$\forall c \neq d,\ S_c.\mathsf{UTXO} \cap S_d.\mathsf{UTXO} = \varnothing$$

Conflict between tx_1 and tx_2 committed by different shards is impossible. There is no conflict among spend-transactions.

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 tx_1 is verified valid only if $tx_1.\mathbf{I}.\mathsf{utxo} \in \mathsf{UTXO}$ However, verified FH-tx tx_2 must satisfy that $tx_2.\mathbf{I}.\mathsf{utxo} \in \mathsf{buffer}$ If $tx_1.\mathbf{I} \cap tx_2.\mathbf{I} \neq \varnothing$, there must be a $\overline{\mathsf{utxo}}$ belong to UTXO and buffer. That is:

$$(tx_1.\mathsf{type} = \textsc{sp}, tx_2.\mathsf{type} = \textsc{fh}) \wedge (tx_1.\mathbf{I} \cap tx_2.\mathbf{I} \neq \varnothing) \iff$$
$$\exists\ \overline{\mathsf{utxo}} \text{ s.t., } \forall c, d,\ \overline{\mathsf{utxo}} \in S_c.\mathsf{UTXO} \cap S_d.\mathsf{buffer}$$

According to Kronos, any transaction output is either added to UTXO or stored in output shard buffer through m_{BF}. No transaction output belongs to UTXO and buffer simultaneously:

$$\forall c, d,\ S_c.\mathsf{UTXO} \cap S_d.\mathsf{buffer} = \varnothing$$

There is no conflict among spend- and finish-transactions.

Consistency among spend- and back-transactions. Similar to FH-txs, the inputs of BK-txs are also spent from shard buffers. While inputs of SP-txs are from UTXOs, the conflict condition is the same as that among spend and finish-transactions:

$$(tx_1.\mathsf{type} = \textsc{sp}, tx_2.\mathsf{type} = \textsc{bk}) \wedge (tx_1.\mathbf{I} \cap tx_2.\mathbf{I} \neq \varnothing) \iff$$
$$\exists\ \overline{\mathsf{utxo}} \text{ s.t., } \forall c, d,\ \overline{\mathsf{utxo}} \in S_c.\mathsf{UTXO} \cap S_d.\mathsf{buffer}$$

Because S_c.UTXO $\cap S_d$.buffer $= \varnothing$ as analyzed above for any shards S_c and S_d, the conflict cannot occur among spend- and back-transactions.

Consistency among finish-transactions. Two conflicting FH-tx tx_1 and tx_2 spend the same $\overline{\text{utxo}}$ from shard buffer. Because each utxo in buffer is identified with the corresponding request's id, the conflict cannot occur among different requests, and the only possible scenario is that $\overline{\text{utxo}}$ is spent for the same request twice by tx_1 committed by BFT$_u$ and tx_2 committed by BFT$_v$. That is:

$$(tx_1.\text{type} = tx_2.\text{type} = \text{FH}) \wedge (tx_1.\mathbf{I} \cap tx_2.\mathbf{I} \neq \varnothing) \iff \exists\ \overline{\text{utxo}} \text{ s.t., } \overline{\text{utxo}} \in S_c.\text{buffer}_u \cap S_c.\text{buffer}_v$$

Because $\overline{\text{utxo}}$ is removed from buffer once its FH-tx is committed. It holds that:

$$\forall u \neq v,\ \ S_c.\text{buffer}_u \cap S_c.\text{buffer}_v = \varnothing$$

Therefore, there is no conflict among finish-transactions.

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. FH-tx$[id]$ is committed only if at least one honest party voted in BFT that all inputs of req$[id]$ have been received and stored in buffer, while BK-tx$[id]$ is committed after receiving $m_{\text{RJ}}[id]$ indicating some input of req$[id]$ is signed unavailable. If both FH-tx tx_1 and BK-tx tx_2 are committed, it holds that:

$$(tx_1.\text{type} = \text{FH}, tx_2.\text{type} = \text{BK}) \wedge (tx_1.\mathbf{I} \cap tx_2.\mathbf{I} \neq \varnothing) \iff \exists\ id \text{ s.t., } \Pr\big((\forall req[id].I,\ I.\text{utxo} \in \text{buffer}_c) \wedge (\exists\, req[id].I \text{ is signed unavailable with } n-f\ m_{\text{RJ}}[id])\big) = 1$$

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. For any cross-shard request $req[id]$, it must hold that:

$$\Pr\big((\forall req[id].I,\ I.\text{utxo} \in \text{buffer}_c) \wedge (\exists req[id].I \text{ is signed unavailable with } n-f\ m_{\text{RJ}}[id])\big) = 0$$

There is no conflict among finish- and back-transactions.

Consistency among back-transactions. The inputs of BK-txs are sourced from buffers, similar to FH-txs. Therefore, the condition of conflicting BK-txs is the same as that of conflicting FH-txs, where

$$(tx_1.\text{type} = tx_2.\text{type} = \text{BK}) \wedge (tx_1.\mathbf{I} \cap tx_2.\mathbf{I} \neq \varnothing) \iff \exists\ \overline{\text{utxo}} \text{ s.t., } \overline{\text{utxo}} \in S_c.\text{buffer}_u \cap S_c.\text{buffer}_v$$

As analyzed before, there is no intersection between S_c.buffer$_u$ and S_c.buffer$_v$ for any shard S_c. Consequently, there is no conflict among back-transactions.

In summary, no conflict transactions are recorded in shard ledgers, where the consistency property is satisfied. ■

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-TRANSACTION.

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 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, which manages 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 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 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 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 SP-tx$[\gamma]$ to spend available inputs. As shards select transactions for BFT from 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 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 FH-tx$[\gamma]$ is constructed after collecting $f+1$ valid signatures from distinct parties and committed by BFT by κ_{BFT} rounds of intra-shard communication. Therefore, the liveness parameter is $\kappa = 2\kappa_{\text{BFT}} + 2$.

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 m_{RJ} are required. The liveness parameter in the happy path is $\kappa = 2$. In unhappy paths, if the 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 BK-tx$[\gamma]$ within another κ_{BFT} rounds of intra-shard communication. In total, the liveness parameter of the unhappy path is $\kappa = 2\kappa_{\text{BFT}} + 1$.

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

APPENDIX E
COMPLEXITY ANALYSIS

Theorem 5 (Optimal intra-shard communication overhead)**.** Kronos *commits a k-shard-involved transaction tx through executing BFT k times totally, 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 2.*

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 consensus, requiring at least one BFT 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 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-ω for valid transaction processing.

In Kronos, input shards spend inputs, and output shards commit transactions, each through a round of BFT, where the IS-ω 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 workload of their output shards because no availability certificate for unavailable inputs can be received. Invalid transactions also do not occupy the BFT 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 the happy path, Kronos processes invalid transactions with the optimal overhead IS-$\omega = 0$ 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-ω for it is $2(x-x')\mathcal{B}$. In Kronos, this worst situation rarely occurs due to fast one-round multicast-based rejection transmission. 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. 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-ω requiring no BFT for invalid transactions in most instances, and there are at most $2(x-x')$ rounds of BFT 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 CS-$\omega = \mathcal{O}(nb\lambda)$, and realizes cross-shard message transmission reliability as per Definition 5.*

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-$\omega_{\text{HT}} = n^2(\mathcal{O}(\frac{b\lambda}{n}) + \mathcal{O}((\log m + \log n)\lambda)) = \mathcal{O}(n(b + n\log m + n\log n)\lambda)$. 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-$\omega_{\text{VC}} = n^2\mathcal{O}(\frac{b\lambda}{n} + \lambda) = \mathcal{O}(nb\lambda)$ 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 F
ARTIFACT APPENDIX

In Section VI, we present results from deploying Kronos on Amazon Web Servers (AWS), scaling from 32 to 1000 nodes across four regions: West Virginia, Hong Kong, Tokyo, and London. We conducted a variety of experiments and comparisons. Except for AHL and ByShard-related data, all data was generated through artifact execution rather than mathematical simulations. The experiments shown in Section VI are not very long-lasting but are implemented on distributed Amazon Web Servers. To meet the artificial evaluation requirements, we provide a scaled-down version to run the experiment with 12 nodes locally.

Cross-shard communication method. In our paper, we propose two optional methods for cross-shard transmission: $\mathcal{O}(n)$-to-$\mathcal{O}(n)$ and $\mathcal{O}(1)$-to-$\mathcal{O}(1)$. In the $\mathcal{O}(n)$-to-$\mathcal{O}(n)$ method, every node in the "sender" shard sends messages to each node in the "destination" shard to ensure reliable communication. In the $\mathcal{O}(1)$-to-$\mathcal{O}(1)$ method, shard leaders are responsible for sending messages, with reliable message transmission ensured through cross-shard view change. This implementation utilizes the $\mathcal{O}(1)$-to-$\mathcal{O}(1)$ transmission method for cross-shard communication, following an intra-shard message broadcast.

Transaction queue. At the beginning of each experiment, transactions that need to be processed are stored by each node. This helps limit the number of execution rounds in a single experiment, reflecting performance within a constrained time frame. Transaction queue size is adjustable.

Throughput measurement. We adopt a conservative approach to throughput calculation. When determining the number of transactions processed in a single intra-shard BFT round, we count the minimum transaction number required to fill a block (i.e., transactions proposed by $2f+1$ nodes rather than by all n nodes, where $n \geq 3f+1$). Therefore, the practical throughput is slightly higher than the test results indicate.

A. Description & Requirements

1) How to access: The artifacts are publicly available on Github[5]. Codes in the `main` branch employ asynchronous BFT protocol, Speeding-Dumbo, for intra-shard consensus, while the `hotstuff` branch contains the version using partially synchronous Rotating-Hotstuff as intra-shard BFT protocol. All necessary instructions for accessing and setting up the artifact are available in the README.md file within the repository. The project is also available on Zenodo[6].

2) Hardware dependencies: No specific hardware dependencies are required for this simulation.

3) Software dependencies: Ubuntu 18.04 LTS is recommended. Other Linux systems can also provide support, as long as the operator can complete the configuration of the environment dependencies. Ensure that python3 is installed.

4) Benchmarks: We adopt a wide-use cross-shard scheme, two-phase commit (2PC), as a benchmark, to measure and evaluate our work's performance on throughput and latency.

[5] https://github.com/XYZ-LABb/sonork

[6] https://doi.org/10.5281/zenodo.13594519

B. Artifact Installation & Configuration

Our repository can be downloaded to local machines via `git clone` command. Installation of all dependencies can be done using the provided `env-batch.sh` script. The dependencies can also be installed manually.

Additionally, Docker containers for easy local experiments are also provided, eliminating the need for dependency installation. The Docker images, `hiddeneer/kronos-1.1` with Speeding-Dumbo and `hiddeneer/kronos-hotstuff-1.0` with Rotating-Hotstuff, have been uploaded to Dockerhub, which can be pulled to local machines via `docker pull` command, and run via `docker run -it` command.

For distributed deployment on AWS, the environment configuration uses `aws-pre.sh` script in the `/aws` directory.

C. Major Claims

- (C1): Kronos is a sharding blockchain consensus framework that ensures security with atomicity and optimized overhead in asynchronous networks. This framework serves as a generic tool for enhancing sharding blockchain performance.
- (C2): Comprehensive performance analysis of Kronos demonstrates its efficiency in terms of throughput, average latency, intra-shard latency, and cross-shard latency, with various network scales, numbers of shards, and transaction batch sizes. This is proven by the experiments, and the results are reported in Section VI-A, VI-B, and Fig. 7, 9, and 10.
- (C3): Successful implementation of interface calls within shards. The framework supports Rotating-Hotstuff and Speeding-Dumbo as intra-shard BFT protocols, allowing for easy integration of other intra-shard protocols for further research. This is proven by the experiments, and the results are reported in Section VI-C and Fig. 11.

D. Evaluation

In this section, we provide scripts for testing and data collection to facilitate both local and AWS-based experiments.

1) Local Experiment Process: The local experiments can be executed with dependency installation or using Docker.

[Preparation] If experimenting locally without Docker, run `env-batch.sh` to install dependencies.

```
sudo ./env-batch.sh
```

If operating in a Docker container, pull the Docker image to local machines. No dependency installation is required.

```
sudo docker pull hiddeneer/kronos-1.1
```

This command pulls the implementation that uses Speeding-Dumbo as an intra-shard BFT protocol. To run the implementation with Rotating-Hotstuff, pull `kronos-hotstuff-1.0` instead as follows.

```
sudo docker pull hiddeneer/kronos-hotstuff-1.0
```

After pulling the image, run the Docker container to start the environment.

```
sudo docker run -it hiddeneer/kronos-1.1
```

Run `hiddeneer/kronos-hotstuff-1.0` instead of `hiddeneer/kronos-1.1` if pulling the version with Rotating-Hotstuff as follows.

```
sudo docker run -it hiddeneer/kronos-hotstuff
    -1.0
```

Upon successfully entering the Docker container, the executing directory should be `kronos`.

[Execution] Run the sharding blockchain consensus.

```
./start.sh [shard-num] [N] [f] [B] [R] (( i *
    node )) node [TXs] [cross-shard TXs]
```

In this command, `shard-num` indicates the number of shards in the system; `N`, `f`, `B`, `R` respectively indicate the shard size, number of Byzantine nodes, BFT batch size and number of BFT rounds run in each experiment. `(( i * node )) node` is used to set which node to start from (usually at 0), `TXs` is the size of each node's transaction queue, and `cross-shard TXs` is the number of overall cross-shard transactions in the system. The last two parameters are useful in adjusting the cross-shard transaction portion to measure the performance in different settings.

A quick start for an experiment is that:

```
./start.sh 3 4 1 1000 5 0 12 1250 3000
```

The settings of this experiment are as follows: the protocol consists of 3 shards, each containing 4 nodes, including 1 Byzantine node, with a BFT batch size of 1000. The experiment spans 5 rounds, with the server running 12 nodes starting from node 0. Each node's transaction queue contains 1250 transactions, and the cross-shard transactions are selected from a pre-generated list of 3000 random cross-shard transactions.

For local testing, only the batch size, transaction pool size, and the number of alternative cross-shard transactions (the 4th, 8th, and 9th parameters) need to be modified. For distributed testing, additional parameter adjustments can be made.

[Results] Assuming that all commands have succeeded with a message "*All nodes finished*", go to the `/data` directory and execute the following Python script to collect and analyze the data:

```
python3 TPS_latency-log.py
python3 delay-log.py
```

When running a machine with 12 virtual cores (hosted by i7-10750H CPU), the throughput for the `main` branch experiment is approximately 1900-2100 tx/s. The throughput for the `hotstuff` branch experiment using Rotating-Hotstuff may exhibit significant fluctuations during local testing. Overall, Kronos with Rotating-Hotstuff tends to achieve higher throughput than with Speeding-Dumbo when the network is not in asynchronous conditions.

Due to the limitations of local device performance, the results only display the throughput and latency data of the three-shard Kronos. Differently from the actual environment, the local experiment environment has limited computing resources but ample communication resources, which may result in outcomes differing from those presented in Section VI, conducted under WAN settings. However, as the number of nodes increases, more and more shards are available, and the performance of the experimental protocol will be significantly improved. In either case, Kronos performs significantly better than the 2PC benchmark.

2) AWS-Based Experiment Process: The `/aws` directory includes scripts for distributed deployment on AWS.

[Preparation] Download the artifact and run `aws-pre.sh` to install dependencies on each server.

```
sudo ./aws/aws-pre.sh
```

[Execution] Run the protocol on AWS servers.

```
sudo ./aws/aws-run
```

[Results] Collect logs from each server, and analyze the collected data similar to local experiments.

```
sudo ./aws/aws-log
python3 TPS_latency-log.py
python3 delay-log.py
```

E. Customization

There are several parameters that can be adjusted in each script, including `start.sh`, `run_socket_node.py`, `tx_generator.py`, etc. Experimenters can change more experimental configurations by modifying the parameters in the scripts, including the proportion of cross-shard transactions, the protocol initiation method (whether it is fully synchronized), and the proportion of valid transactions, etc.