Breaking Blockchain's Communication Barrier with Coded Computation

Canran Wang and **Netanel Raviv**
Department of Computer Science and Engineering, Washington University in St. Louis, St. Louis, MO 63103.

arXiv:2204.00979v1 [cs.DC] 3 Apr 2022

Abstract

Although blockchain, the supporting technology of various cryptocurrencies, has offered a potentially effective framework for numerous decentralized trust management systems, its performance is still sub-optimal in real-world networks. With limited bandwidth, the communication complexity for nodes to process a block scales with the growing network size and hence becomes the limiting factor of blockchain's performance.

In this paper, we suggest a re-design of existing blockchain systems, which addresses the issue of the communication burden. First, by employing techniques from Coded Computation, our scheme guarantees correct verification of transactions while reducing the bit complexity dramatically such that it grows logarithmically with the number of nodes. Second, with the adoption of techniques from Information Dispersal and State Machine Replication, the system is resilient to Byzantine faults and achieves linear message complexity. Third, we propose a novel 2-dimensional sharding strategy, which inherently supports cross-shard transactions, alleviating the need for complicated communication protocols between shards, while keeping the computation and storage benefits of sharding.

I. Introduction

Blockchain is an append-only decentralized system, in which data resides in a chain of blocks that are periodically proposed and agreed upon by a consensus mechanism. Although it is a promising platform for various applications, its performance is sub-optimal due to the limited bandwidth and the scaling communication complexity, in terms of *message complexity* and *bit complexity*. Message complexity is measured by the number of transferred messages, and bit complexity is characterized by the number of communicated bits.

The performance of Bitcoin [2], the first double-spending-resistent cryptocurrency in a public peer-to-peer network, is inherently limited by its design. For a valid new block to be generated, the competing nodes invest the majority of time in solving Proof-of-Work (PoW) puzzles. Consensus is reached on the sole block proposed by the winner, which is then propagated to the remaining nodes and appended to each local chain. Such a concatenated consensus-then-propagation scheme fails to fully utilize the bandwidth of nodes, since the network remains idle during the PoW puzzle solving period.

Meanwhile, the security of Bitcoin is guaranteed by the fact that the time interval between blocks is sufficiently greater than the block transmission time [3]. Otherwise, frequent forks, which occur when multiple blocks are proposed simultaneously cause temporary inconsistency between nodes, and jeopardize the safety of the system. In other words, the PoW puzzle should take a sufficiently long period of time to solve, compared with the required time for the majority of node to receive a block. Together, naïve reparameterization such as reducing the difficulty of the PoW puzzle or enlarging the block size degrades security, and a comprehensive redesign is required to improve Bitcoin's performance.

A widely adopted paradigm to achieve high-performance blockchain systems is to parallelize consensus and propagation, and hence to maximize the efficiency of bandwidth usage [4, 5, 6, 46]. Works following this path inherit the PoW mechanism to periodically select an entity from the public network as a leader, which could be a node or a committee of nodes. The selected entity is allowed to continuously generate blocks in parallel with the leader election mechanism, until the next entity is selected. Compared with Bitcoin, this paradigm persistently utilizes the bandwidth of nodes, and hence improves system performance.

In another direction, researchers attempt to improve Bitcoin by replacing its PoW mechanism, which is seen as the root cause of the scalability issue and huge energy consumption. Proof-of-Stake (PoS) is a noteworthy alternative used by [7, 8, 9], which does not involve the computation-intensive PoW puzzle solving. Instead, the chance for each individual node being selected as the leader, or one of the leaders, is proportional to its *stake*, referring to the value resides in the blockchain system.

Although the aforementioned attempts improve the performance of blockchain to some extent, a fundamental obstacle remains. That is, *every node must receive every transaction*. This requirement is paramount to the safety and decentralization of blockchain systems, but unfortunately leads to an inevitable $\Omega(NP)$ bit complexity for a block $\mathbf{B}$ containing P transactions to be confirmed, given a network of N nodes.

Sharding [23] is a novel paradigm proposed to address this problem. The network is sliced into multiple communities of a similar sizes, each individually processes a disjoint set of data [24, 25, 26]. The constant community size reduces the communication complexity as one transaction is only propagated within one community. As a result, the system throughput scales with the number of nodes, as additional nodes form extra communities and process additional transactions.

Parts of this paper have previously appeared in [1].

In sharding-based blockchain designs, random node rotation, or even reassignment, is necessary to avoid concentration of adversaries in one community. Further, sharding creates a distinction between two types of transactions; a transaction is called intra-shard if the sender and the receiver belong to the same community, and called cross-shard otherwise. Hence, extra mechanisms are required in this path, which is an added complexity that degrades the system's performance and diminishes the benefits of sharding.

Coding has been introduced to bypass the requirement for every node to receive every transaction, which leads to the invention of *coded blockchain*. Duan *et al.* propose *BEAT3* [27], a BFT storage system that enables each node to periodically store a relatively small coded fragment generated from the whole data block. The error-correcting code guarantees reconstruction of the original data block from sufficiently many of fragments. The AVID-FP [32] protocol is used to assure that fragments stored by correct nodes correspond to a unique original data block. However, as a BFT storage system, BEAT3 does not concern *external validity*, which assures that the stored data is acceptable to a specific application [28]. In Blockchain's scenario, nodes in BEAT3 cannot verify the correctness of each stored transaction.

The introduction of coded computation partially alleviates the security problems in sharding, and provides support for external validity. Li *et al.* [29] proposed *Polyshard*, which offers a novel separation between nodes and shards. Polyshard formulates the verification of transactions as computation tasks, one for each shard, to be solved across all nodes in a distributed manner. Using Lagrange Coded Computing (LCC) [30], nodes individually compute a polynomial verification function over a *coded chain* and a *coded block*. Since verification is performed in a coded fashion, and a node does not verify or store transactions for any specific shard, the need for node rotation/reassignment is removed.

Polyshard implicitly assumes that the performance bottleneck stems from insufficient computation resources in nodes, rather than limited communication bandwidth, and considers the system as a computation cluster with a highly synchronous network. The bit complexity, however, is again $O(NP)$, as Polyshard requires every node to firstly reach a consensus of the whole block $\mathbf{B}$ and then perform encoding individually. Otherwise, as pointed out in [11], the system can be broken by a discrepancy attack. Besides, the messages complexity is $O(N^2)$, due to the fact that Polyshard involves an all-to-all communication operation.

Finally, coding has been employed in blockchain system that allows *light nodes*. Unlike *full nodes* that validate and store all transactions, light nodes only download the header of each block, and hence addressing the $O(NP)$ bit complexity. The header contains the root of a Merkle tree whose transactions are the leaves; it allows light nodes to verify the inclusion of any transaction in the corresponding block by downloading a Merkle proof from full nodes. However, the *data availability problem* arises, i.e., upon receiving a header and Merkle proof, a light node cannot assure the corresponding block is fully available to the network, while the undisclosed part of the block may be invalid. A coding-based solution to data availability problem has bee proposed by [47] and further improved by [48]. In this paper, we only consider full nodes, and leave the incorporation of light nodes for future work.

Our Contributions

In this paper, we propose a fundamental re-design for coded blockchain, which resolves many of the issues in contemporary coded blockchain systems. In particular, this re-design addresses the issue that every node should receive every transaction, and hereby resolves the presumably inevitable $\Omega(NP)$ bit complexity, that is also prevalent in ordinary (that is, uncoded) blockchain systems. Further, it achieves linear message complexity by resolving the issue of all-to-all communication, which is message-heavy but necessary for decoding the results of the computation. On top of this gain, we adopt Lagrange coded computing—similar to existing designs—to achieve comparable levels of decentralization and security guarantees with respect to uncoded (i.e., ordinary) blockchain. In detail,

1) By employing techniques from Lagrange Coded Computing [30], our scheme allows nodes to perform verification on *coded transactions*, whose size is a fraction of the entire block. Our method incurs $O(P \log^2 M \log N)$ bit complexity to process a block with P transactions, where M is the total number of transactions in one shard. As an alternative interpretation, the average bit complexity to process a block is $O(\log^2 M \log N)$.
2) By devising techniques inspired by Information Dispersal and BFT SMR protocols, our design allows a leader node to securely distribute coded transactions, under the presence of a certain fraction of Byzantine nodes, with $O(N)$ message complexity in the partial synchrony model. In the suggested parameter regime and under standard cryptographic assumptions, our design is provably secure to any attack that aims at breaking the consistency of the system, and in particular the attack pointed out by [11].
3) We propose *2-Dimensional Sharding*, a new technique which partitions the transactions based on their senders and receivers, respectively. This design provides inherent support for cross-shard transactions, alleviating the need for complicated communication mechanisms. More precisely, in our design there is *no difference* between the verification process of cross- and intra-shard transactions.
4) Our design inherits the *unspent transaction output* (UTXO) model and formulates the verification process as computing a polynomial function with a degree that scales logarithmically with the number of transactions on a shard. In detail, our scheme addresses the degree problem by replacing current cryptographic primitives (i.e, ECDSA, SHA256 and RIPEMD-160) by multivariate cryptography in the generation and verification of a transaction.

These contributions bring coded blockchain closer to feasibility. That is, our scheme achieves linear message complexity and logarithmic bit complexity and removes the boundary between shards with inherent support for cross-shard transactions. The rest of this paper is organized as follows. Section II introduces necessary background. Section III details the coded verification scheme. Section IV discusses the communication aspect of our design, including the propagation of transactions and the exchange of computation results. Section V analyzes the security, communication complexity, and the tradeoff between them. Section VI discusses the future research directions.

II. Background

A. Lagrange Coded Computing

Coded computing broadly refers to a family of coding-inspired solutions for straggler- and adversary-resilient distributed computation. Tasks of interest include matrix-vector multiplication [51], matrix-matrix multiplication [52], gradient-computations [53, 54], and more. Further works on the topic include exploitation of partial stragglers [55], heterogeneous networks [57], and timely coded computing [56].

Lagrange Coded Computing [30] (LCC) is a recent development in the field of coded computation. The task of interest is computing a multivariate polynomial $f(X)$ on each of the K datasets $\{X_1, \dots, X_K\}$. LCC employs the Lagrange polynomial to linearly combine the K datasets with T redundant datasets $\{Z_1, \dots, Z_T\}$ chosen uniformly at random, generating N distinct coded dataset $\{\widetilde{X}_1, \dots, \widetilde{X}_N\}$ with injected computational redundancy.

The encoding of LCC is performed by first choosing mutually disjoint sets $\{\alpha_1, \dots, \alpha_N\}$ and $\{\omega_1, \dots, \omega_K, \dots, \omega_{K+T}\}$ with elements in $\mathbb{F}_q$. The generator matrix $G_{\mathcal{L}}$ is then defined as

$$G_{\mathcal{L}} = \begin{bmatrix} \Phi_1(\alpha_1) & \Phi_1(\alpha_2) & \dots & \Phi_1(\alpha_N) \\ \vdots & \vdots & \ddots & \vdots \\ \Phi_{K+T}(\alpha_1) & \Phi_{K+T}(\alpha_2) & \dots & \Phi_{K+T}(\alpha_N) \end{bmatrix}, \tag{1}$$

where $\Phi_k(z)$ is the Lagrange polynomial

$$\Phi_k(z) = \prod_{j,k \in [K+T], j \neq k} \frac{z - \omega_j}{\omega_k - \omega_j}. \tag{2}$$

The coded datasets is generated as $(\widetilde{X}_1, \dots, \widetilde{X}_N) = (X_1, \dots, X_K, Z_1, \dots, Z_T) \cdot G_{\mathcal{L}}$. Every worker node $i \in [N]$ computes and returns a coded result $f(\widetilde{X}_i)$. The leader obtains $f(X_1), \dots, f(X_K)$ by performing decoding on collected coded results.

LCC achieves the optimal tradeoff between resiliency, security and privacy. It tolerates up to S stragglers and A adversarial nodes, defined as working nodes that are unresponsive or return erroneous results, respectively. In addition, with proper incorporation of random keys, it also prevents the exposure of the original datasets to sets of at most T colluding workers, as long as

$$(K + T - 1) \deg\ f + S + 2A + 1 \leq N.$$

B. State Machine Replication

The state machine replication (SMR) approach [12, 13] formulates a service, e.g., a network file system, as a state machine to be replicated in participating nodes. The state can be altered by client-issued service *requests*. To ensure the consistency of the states, nodes must agree on a total order of execution for requests.

An SMR implementation must guarantee *safety* and *liveness*. Safety suggests that no two nodes confirm different order of requests, and *liveness* imposes that the system continuously accepts and executes new requests. Further, an SMR protocol is said to be Byzantine Fault-Tolerant (BFT) if it is resilient to Byzantine faults, as coined by Lamport *et al.* [10], which are defined as arbitrary (and possibly malicious) behaviour of nodes.

Network models plays an important role in the design of SMR protocols. In an asynchronous systems, message can be delayed by any finite amount of time, but eventual delivery is guaranteed. BFT SMR protocols which operate in this model employ randomization to bypass the famous FLP impossibility [50]. This impossibility result states that in the presence of even one faulty node (not necessarily Byzantine), it is impossible to guarantee consensus with a deterministic protocol. Works following this path include [21, 22, 27].

As proposed by Dwork *et al.* [49], partial synchrony is another noteworthy network model. In this setting, message delivery is asynchronous until an unknown Global Stabilization Time (GST). After GST, the system becomes synchronous, where message delay is bounded by a known constant Δ. PBFT [15] is the first practical implementation of BFT SMR in the the partial synchrony model. It guarantees safety always, and provides liveness when the system becomes synchronous.

PBFT employs a leader to propose client-issued requests, and it takes two phases of all-to-all communication for the decision on one request. To prohibit Byzantine leaders from proposing different requests to different nodes, a proposal is considered valid only after being signed by a quorum of $N - f$ nodes in the first phase, known as a *quorum certificate (QC)*, where f is the

number of Byzantine nodes. Next, nodes commit the request after receiving another $N-f$ votes in the second phase. A quorum contains enough nodes such that any two quorums must intersect on at least one correct node [14]. Such a property guarantees that correct nodes entering the second phase are consistent on the same request. In addition, it assures that the proposals from subsequent leaders (should the previous one crash) are consistent in request and hence maintains safety during leader switches. This celebrated two-phase paradigm serves as the foundation of future leader-based BFT SMR protocols [16, 17, 18, 19].

Bitcoin coined the word blockchain, providing an alternative implementation of SMR, particularly for value transfer systems in large networks. It maintains an ordered sequence (chain) of blocks (requests), each contains transactions that incur value (bitcoins) transfers between clients. Nodes invest computation power into PoW puzzle solving for the right to propose the next block; they are incentived by a reward in values. In particular, nodes look for a new block by trial and error. The new block must extends the current chain (i.e., contains a hash pointer to the last block on the chain), and the hash value of which must satisfy a certain rule (e.g., begin with a sequence zeros). The winner of the competition disseminates its block to the network by gossip protocol, which is then appended to each local chain. Unlike the protocol discussed earlier, the safety of Bitcoin relies on synchrony. Further, the *finality* property (i.e., a consensus once reached cannot be reverted) of Bitcoin is probabilistic. In practice, a block is considered irrevertible after being followed by six new blocks. Numerous blockchain designs have been introduced to improve Bitcoin (see Section I).

HotStuff [20] bridges PBFT-like protocols and Bitcoin-like protocols. It extends the two-phase paradigm of PBFT to three phases, each contains a nearly identical communication operation between the leader and the nodes. Due to this remarkable simplicity, HotStuff can be easily pipelined; i.e., the second phase on a block functions as the first phase on the following block, as well as the last phase on the preceding block. Therefore, a block is irrevitible after three new blocks being appended to it, which is similar to the case of Bitcoin. Besides, the extra phase allows HotStuff to be the first protocol that simultaneously achieves linear message complexity and *responsiveness* during leader switches. A leader switch is said to be responsive if the new leader only has to collect a quorum of leader-switch messages, and there is no requirement for it to wait for a predefined time period.

Due to these merits, we adopt HotStuff as the core consensus protocol, and employ techniques from coded computation and information dispersal (define next) to reduce bit complexity. Our work can be regarded as a communication-efficient implementation of coded state machine [58], which simultaneously maintains K state machines (shards) and employ coded computation to combat Byzantine faults.

Remark 1. BFT SMR protocols focus on the communication complexity induced by reaching a consensus on the order of the requests. It is generally assumed that each request is broadcast to every node by the issuing client, and this process is out of the scope of communication complexity analysis. However, blockchain systems usually require the leader to collect and distribute transactions, which must be considered in analyzing the communication complexity.

C. Information Dispersal

In a coded distributed information system, a file $X \in \mathbb{F}_q^{\delta K}$ to be stored is first partitioned to K parts $X = (X_1^\intercal, \ldots, X_K^\intercal)$ where $X_k^\intercal \in \mathbb{F}_q^{\delta \times 1}$. A Maximum Distance Separable (MDS) error-correcting code $\mathcal{C}$, induced by a *generator matrix* $G_\mathcal{C} \in \mathbb{F}_q^{K \times N}$, is used to generate N coded fragments $\widetilde{X} = (\widetilde{X}_1^\intercal, \ldots, \widetilde{X}_N^\intercal) = X \cdot G_\mathcal{C}$. Each of the N nodes stores one coded fragment. The MDS property of $\mathcal{C}$ codes guarantees that any $K \times K$ submatrix of $G_\mathcal{L}$ is of full rank and hence any K coded fragments are sufficient to reconstruct X, tolerating up to $N-K$ crashes.

Research in this field normally concerns a scenario where an external client wants to *disperse* a file X to the system. That is, for every node i to store the corresponding coded fragment $\widetilde{X}_i$. Byzantine faults can cause inconsistency of coded fragments, i.e., nodes might store coded fragments that do not correspond to the same file X. Efforts has been made on developing protocols to combat Byzantine faults in this scenario.

AVID-FP (where FP stands for *fingerprinting*) [32] enables a client to distribute coded fragments of some file X to nodes in a distributed system, along with a checksum, i.e., a list of *fingerprints* of every coded fragment. AVID-FP inherits the properties of Cachin's Asynchronous Verifiable Information Dispersal (AVID) protocol [31], with additional fingerprints. The fingerprints, generated by a homomorphic fingerprinting function (defined formally in the sequel) preserves the structure of error-correcting codes, and allows node i to verify that the received fragment corresponds to a unique file X. In this paper, we propose an efficient transaction propagation scheme that integrates the steps of AVID-FP and coding techniques (see Section IV).

D. The Unspent Transaction Output (UTXO) Model

In the UTXO model, value resides in transactions, instead of client accounts. A transaction has inputs and outputs. An unspent output of an old transaction serves as an input to a new transaction, incurring a value transfer between the two. The old UTXO is then invalidated, since it has been spent, and new UTXO is created in the new transaction.

The UTXO model makes extensive use of cryptographic hash functions and digital signatures. The uninformed reader is referred to [2, Sec. 2] for a thorough introduction to the topic. In a nutshell, a transaction output contains the amount of stored value and the intended receiver's *address*, which is the hash value of her public key. Besides, the sender attaches his public

key and signs the transaction with his secret key. For a transaction to be valid, the hash value computed from the sender's public key must match the address in the referenced UTXO. Also, the signature must be valid when checked by the public key. This two-step verification process guarantees the sender's possession of the public and secret keys, proves his identity as the receiver of the redeemed UTXO, and protects the integrity of the new transaction.

Although a transaction may have multiple inputs and outputs, we adopt a simplified UTXO model in our scheme for clarity, where a transaction has exactly one input and one output, and transfers one indivisible coin.

E. Cryptographic Primitives

We assume that a public key infrastructure (PKI) exists among nodes. That is, every node i can create a signature $\langle m \rangle_{\sigma_i}$ on a message m using its private key σ_i. Meanwhile, such a signature can be verified by the corresponding public key, which is shared by all nodes. Further, we employ a *threshold signature* [34] scheme. A (t, n)-threshold signature scheme π contains a single public key shared by all nodes. Every node i possess a private key π_i which allows it to create a *partial signature* $\langle m \rangle_{\pi,i}$ on message m. A valid threshold signature $\langle m \rangle_\pi = \textit{tcombine}(m, \{\langle m \rangle_{\pi,i}\}_{i \in \mathcal{I}})$ can be produced using function *tcombine* from a set of partial signatures $\{\langle m \rangle_{\pi,i}\}_{i \in \mathcal{I}}$ of size $|\mathcal{I}| = t$, but not smaller. Hence, it is guaranteed that the message m has been signed by t nodes if the signature verification function $\textit{tverify}(m, \langle m \rangle_\pi)$ returns true.

In addition, in order to formulate the verification of transactions as the computation of polynomials, clients use a multivariate public key cryptosystem (MPKC) [36, 37, 38] as a signature scheme. MPKC is based on the multivariate quadratic (MQ) problem, which is believed to be hard even for quantum computers. An MQ problem involves a system of m quadratic polynomials $\{p^{(1)}, \dots, p^{(m)}\}$ in n variables $\{y_1, \dots, y_n\}$ over some finite field $\mathbb{F}_q$, i.e.,

$$\mathbf{p}(\mathbf{y}) = \sum_{0<i\leq j<n} \mathbf{a}_{(i,j)} y_i y_j + \sum_{0<i<n} \mathbf{b}_i y_i + \mathbf{c},$$

where $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ are vectors in $\mathbb{F}_q^m$. The solution is a vector $\mathbf{u} = (u_1, \dots, u_n) \in \mathbb{F}^n$ such that $\mathbf{p}(\mathbf{u}) = (0, \dots, 0) \in \mathbb{F}_q^m$.

In general, the public key of a MPKC is the set of coefficients of the quadratic polynomial system. A valid signature $\mathbf{s} \in \mathbb{F}_q^n$ on a message $\mathbf{w} \in \mathbb{F}_q^m$ is the solution to the quadratic system $\mathbf{p}(\mathbf{y}) = \mathbf{w}$. In addition to MQ-based signature schemes, hash functions based on multivariate polynomials of low degree have been studied [40, 41, 42]. In the remainder of this paper, we assume a polynomial hash function over $\mathbb{F}_q$ of a constant degree.

III. Coded Verification

In this section, we first introduce our general settings and assumptions. Based on these settings, we discuss the verification of transactions. As in the UTXO model, the verification process starts from fetching an existing transaction stored in the chain, and proceeds with the address check and signature verification process. Together, the entire verification is formulated as computing a polynomial function. Consequently, we demonstrate the incorporation of Lagrange Coded Computing, showing how verification can be performed in a coded manner.

A. Setting

The system includes N nodes and K client communities of equal size. The nodes are responsible for collecting, verifying and storing transactions; clients issue transactions and transfer coins between each other. Note that clients are affiliated with communities, whereas nodes are not. Transactions are proposed by clients and verified by nodes periodically during time intervals, called *epochs*, denoted by a discrete time unit t.

We formulate the block containing all transactions in epoch t as a matrix,

$$\mathbf{B}^{(t)} = \begin{bmatrix} b_{1,1} & b_{1,2} & \dots & b_{1,K} \\ \vdots & \vdots & \ddots & \vdots \\ b_{K,1} & b_{K,2} & \dots & b_{K,K} \end{bmatrix}, \tag{3}$$

where every $b_{k,r} \in \mathbb{F}_q^{Q \times R}$ is a *tiny block*, formed as a concatenation of Q transactions with senders in community k and receivers in community r; each transaction $\mathbf{x} \in \mathbb{F}_q^R$ is a vector of length R over some finite field $\mathbb{F}_q$.

We partition the block $\mathbf{B}^{(t)}$ into *outgoing strips* and *incoming strips*, as shown in Fig. 1. An outgoing strip

$$\mathbf{h}_k^{(t)} = (b_{k,1}, \dots, b_{k,K}) \in (\mathbb{F}_q^{Q \times R})^K$$

is a vector containing transactions with senders in community k. Similarly, an incoming strip

$$\mathbf{v}_k^{(t)} = (b_{1,k}, \dots, b_{K,k}) \in (\mathbb{F}_q^{Q \times R})^K$$

stands for a collection of all transactions in epoch t with receivers in community k. Equivalently, one can view an outgoing strip $\mathbf{h}_k^{(t)}$ as the k-th row of matrix $\mathbf{B}^{(t)}$, and incoming strip $\mathbf{v}_k^{(t)}$ as the transpose of the k-th column of matrix $\mathbf{B}^{(t)}$, i.e.,

$$\mathbf{B}^{(t)} = \left[(\mathbf{v}_1^{(t)})^\intercal, (\mathbf{v}_2^{(t)})^\intercal, \dots, (\mathbf{v}_K^{(t)})^\intercal \right] = \left[(\mathbf{h}_1^{(t)})^\intercal, \quad (\mathbf{h}_2^{(t)})^\intercal, \quad \dots, \quad (\mathbf{h}_K^{(t)})^\intercal \right]^\intercal.$$

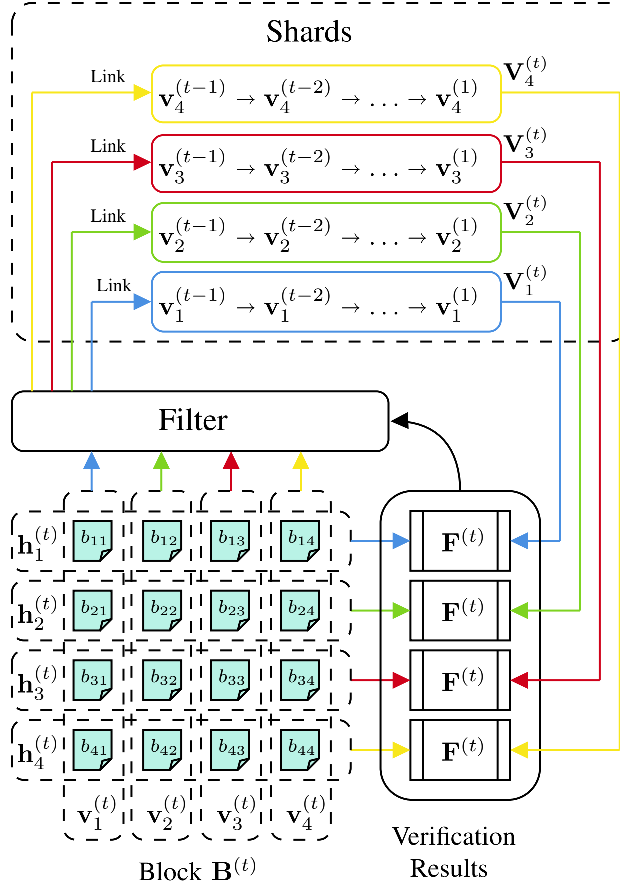

Fig. 1. Illustration of *2-Dimensional Sharding* in a blockchain system with 4 shards. The block $B^{(t)}$ is horizontally sliced into *outgoing strips* $\mathbf{h}_1^{(t)}, \ldots, \mathbf{h}_4^{(t)}$ and vertically sliced into *incoming strips* $\mathbf{v}_1^{(t)}, \ldots, \mathbf{v}_4^{(t)}$. The outgoing strip $\mathbf{h}_k^{(t)}$ is verified against the corresponding shard $\mathbf{V}_k^{(t)}$ using the verification function $\mathbf{F}^{(t)}$, for $k = 1, 2, 3, 4$. Together, the verification results reveal the validity of every transaction, and help to filter out the invalid transactions in the incoming strips, which are finally linked to the corresponding shards.

Formally, we define a *shard* $\mathbf{V}_k^{(t)} = \left(\mathbf{v}_k^{(1)}, \ldots, \mathbf{v}_k^{(t)}\right)$ as a concatenation of *incoming* strips associated with community k from epoch 1 to epoch t, which contains $M(t)$ transactions. Note that our definition of a shard is slightly different from the existing literature[1]. This definition provides a inherent support for cross-shard transactions, which will be elaborated in sequel.

Remark 2. A coded outgoing strip $\mathbf{h}_k^{(t)}$ contains transactions redeeming UTXOs from shard $\mathbf{V}_k^{(t)}$. In later sections, we present a polynomial function that verifies $\mathbf{h}_k^{(t)}$ against $\mathbf{V}_k^{(t)}$. The results are used to filter out invalid transactions in the incoming strip $\mathbf{v}_k^{(t)}$ before they are appended to the shard. As a result, our setting does not differentiate intra- and cross-shard transactions, alleviating the need for sophisticated cross-shard communication mechanisms.

B. Polynomial Verification Function

In our setting, a new transaction is of the form $\mathbf{x}_{new} = (\mathbf{u}_{new}, \mathbf{p}_{new}, \mathbf{a}_{new}, \mathbf{s}_{new})$, where:

1) $\mathbf{u}_{new} \in \mathbb{F}_q^{T^{(t-1)} \times 2}$ is a *lookup matrix* used to index the previous transaction, where $T^{(t)} = \log_2 M(t)$.
2) $\mathbf{p}_{new} \in \mathbb{F}_q^{B}$ is the sender's public key, containing all coefficients of an MQ system.
3) $\mathbf{a}_{new} \in \mathbb{F}_q^{C}$ is the receiver's address, i.e., the hash value of the receiver's public key.
4) $\mathbf{s}_{new} \in \mathbb{F}_q^{D}$ is the senders signature on $\mathbf{x}'_{new} = (\mathbf{u}_{new}, \mathbf{p}_{new}, \mathbf{a}_{new})$

Verifying $\mathbf{x}_{new}$ includes three crucial parts:

- **Transaction Fetching**: To fetch the corresponding old transaction $\mathbf{x}_{old} = (\mathbf{u}_{old}, \mathbf{p}_{old}, \mathbf{a}_{old}, \mathbf{s}_{old})$ from which $\mathbf{x}_{new}$ redeems the UTXO.
- **Address Checking**: To check whether the hash value of $\mathbf{p}_{new}$ matches $\mathbf{a}_{old}$.
- **Signature Verification**: To verify that $\mathbf{s}_{new}$ is a valid signature on the hash value of $\mathbf{x}'_{new}$ by using the public key $\mathbf{p}_{new}$.

In detail, the above parts are executed as follows.

[1]Sharding in blockchain broadly refers to the practice of partitioning nodes among different committees (in a possibly random fashion), each individually handles a portion of verification and storage [23]. On the contrary, we do not assign individual node to any specific committee, but partition clients among communities. Meanwhile, we partition transactions based on the community of the receivers, and each partition is called a shard.

1) Transaction Fetching: The lookup matrix $\mathbf{u}_{new}$ has exactly one 1-entry and one 0-entry in each row. Hence, every transaction in $\mathbf{V}^{(t)}$ can be uniquely indexed by a lookup matrix. The verifier views shard $\mathbf{V}^{(t)}$ as $T^{(t-1)}$-dimensional tensor in $(\mathbb{F}_{q^R})^{2\times 2\times\dots\times 2}$, and therefore every transaction can be conveniently expressed as a tensor entry $\mathbf{V}^{(t)}_{i_1,\dots,i_{T^{(t-1)}}} \in \mathbb{F}_{q^R}$.

To fetch a transaction, one computes a multilinear polynomial,

$$\textit{fetch}^{(t)}(\mathbf{u}, \mathbf{V}^{(t)}) = \sum_{(i_1,\dots,i_{T^{(t-1)}})\in\{1,2\}^{T^{(t-1)}}} \left(\prod_{j=1}^{T^{(t-1)}} \mathbf{u}_{j,i_j}\right) \mathbf{V}^{(t)}_{i_1,\dots,i_{T^{(t-1)}}}$$

which takes a shard $\mathbf{V}^{(t)}$ and a lookup table $\mathbf{u}$ as inputs and yields the transaction $\mathbf{x_u} \in \mathbb{F}_{q^R}$ indexed by $\mathbf{u}$. The degree of $\textit{fetch}^{(t)}$ is $T^{(t-1)}+1$. Note that the subscript k is omitted in $\textit{fetch}^{(t)}$ since it can be applied to any shard.

Example 1. *In a shard* $\mathbf{V}$ *that contains* 8 *transactions, to fetch one of them, one would compute*

$$\begin{aligned}\textit{fetch}(\mathbf{u}, \mathbf{V}) = {} & \mathbf{u}_{1,1}\mathbf{u}_{2,1}\mathbf{u}_{3,1}\ \mathbf{V}_{1,1,1} + \mathbf{u}_{1,1}\mathbf{u}_{2,1}\mathbf{u}_{3,2}\ \mathbf{V}_{1,1,2} + \mathbf{u}_{1,1}\mathbf{u}_{2,2}\mathbf{u}_{3,1}\ \mathbf{V}_{1,2,1} + \mathbf{u}_{1,1}\mathbf{u}_{2,2}\mathbf{u}_{3,2}\ \mathbf{V}_{1,2,2} \\ & + \mathbf{u}_{1,2}\mathbf{u}_{2,1}\mathbf{u}_{3,1}\ \mathbf{V}_{2,1,1} + \mathbf{u}_{1,2}\mathbf{u}_{2,1}\mathbf{u}_{3,2}\ \mathbf{V}_{2,1,2} + \mathbf{u}_{1,2}\mathbf{u}_{2,2}\mathbf{u}_{3,1}\ \mathbf{V}_{2,2,1} + \mathbf{u}_{1,2}\mathbf{u}_{2,2}\mathbf{u}_{3,2}\ \mathbf{V}_{2,2,2}.\end{aligned}$$

Since the lookup matrix contains only one 1-entry and one 0-entry, only the entry indexed by $\mathbf{u}$ *has coefficient* 1*, while the rest have coefficients* 0*.*

2) Address Checking: Based on Section II-E, we assume a multivariate polynomial $\textit{hash1} : \mathbb{F}_q^B \to \mathbb{F}_q^C$ of a constant degree to serve as our first collision resistant hash function. Having obtained $\mathbf{x}_{old} = \textit{fetch}^{(t)}(\mathbf{u}_{new}, \mathbf{V}_k^{(t)})$, the verifier then checks whether $\textit{hash1}(\mathbf{p}_{new}) = \mathbf{a}_{old}$ holds, which is expressed as a polynomial,

$$\textit{checkAddr}(\mathbf{p}, \mathbf{a}) = \textit{hash1}(\mathbf{p}) - \mathbf{a}.$$

Note that $\mathbf{p}_{new}$ is accepted when $\textit{checkAddr}(\mathbf{p}_{new}, \mathbf{a}_{old}) \in \mathbb{F}_q^C$ is the all-zero vector.

3) Signature Verification: The verifier needs to check the validity of the signature $\mathbf{s}_{new}$. She first computes a hash digest $\mathbf{w} = \textit{hash2}(\mathbf{u}_{new}, \mathbf{p}_{new}, \mathbf{a}_{new}) = (w_1, \dots, w_E) \in \mathbb{F}_q^E$, where $\textit{hash2} : \mathbb{F}_q^{A+B+C} \to \mathbb{F}_q^E$ is our second collision resistant hash function of a constant degree. Later, the verifier checks whether $MQ(\mathbf{p}_{new}, \mathbf{s}_{new}) = \mathbf{w}$ holds, where,

$$MQ(\mathbf{p}, \mathbf{s}) = \sum_{0<i\leq j<D} \mathbf{a}_{(i,j)} s_i s_j + \sum_{0<i<D} \mathbf{b}_i s_i + \mathbf{c},$$

and $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{F}_q^E$ are vectors stored in $\mathbf{p}_{new}$, serving as coefficients of the MQ problem. Equivalently, the verification of a signature $\mathbf{s}$ in a transaction $\mathbf{x} = (\mathbf{u}, \mathbf{p}, \mathbf{a}, \mathbf{s})$ can be expressed as a polynomial,

$$\textit{checkSig}(\mathbf{x}) = MQ(\mathbf{p}, \mathbf{s}) - \textit{hash2}(\mathbf{u}, \mathbf{p}, \mathbf{a}).$$

Note that $\mathbf{s}_{new}$ is accepted only when $\textit{checkSig}(\mathbf{x}_{new}) = 0$.

The above three parts focus on the verification of an individual transaction. We further employ them to verify the entire strip as follows.

4) Verification of a strip: Let $\eta \in \mathbb{F}_q^{C+E}$ be the concatenation of $\textit{checkAddr}(\mathbf{p}_{new}, \mathbf{a}_{old})$ and $\textit{checkSig}(\mathbf{x}_{new})$; a transaction $\mathbf{x}$ is accepted if and only if $\eta = f^{(t)}(\mathbf{x}, \mathbf{V}^{(t)}) = 0$. This information is further used to fliter out invalid trancations in the incoming strip. Since the UTXOs redeemed by transactions in $\mathbf{h}_k^{(t)} = (\mathbf{x}_1, \dots, \mathbf{x}_{QK})$ all reside in $\mathbf{V}_k^{(t)}$, we define a multivariate polynomial,

$$F^{(t)}(\mathbf{h}^{(t)}, \mathbf{V}^{(t-1)}) = (f^{(t)}(\mathbf{x}_1, \mathbf{V}^{(t-1)})^\intercal, \dots, f^{(t)}(\mathbf{x}_{QK}, \mathbf{V}^{(t-1)})^\intercal),$$

of degree d, which yields an *outgoing result strip*

$$\mathbf{e}_k^{(t)} = (r_{k,1}, \dots, r_{k,K}) \in (\mathbb{F}_q^{Q\times(C+E)})^K, \tag{4}$$

defined as the k-th row of the *result matrix*

$$\mathbf{R}^{(t)} = \begin{bmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,K} \\ \vdots & \vdots & \ddots & \vdots \\ r_{K,1} & r_{K,2} & \cdots & r_{K,K} \end{bmatrix}. \tag{5}$$

Each *tiny result block* $r_{k,k'}$ contains Q entries of length $C+E$; one per every transaction in the tiny block $b_{k,k'}$. The j-th entry in $r_{k,k'}$ is the result of computing $f^{(t)}$ on $\mathbf{V}_k^{(t)}$ and the j-th transaction in $b_{k,k'}$. Hence, the outgoing result strip $\mathbf{e}_k^{(t)}$ reveals the validity of every transaction in the outgoing strip $\mathbf{h}_k^{(t)}$, and the result matrix $\mathbf{R}^{(t)}$ reveals the validity of every transaction in the block $\mathbf{B}^{(t)}$. As shown in Fig. 1, the outgoing result strips are used to filter out the invalid transactions in the incoming strips before they are being appended to the corresponding shards.

Similarly, the *incoming result strip*

$$\mathbf{s}_k^{(t)} = (r_{1,k}, \ldots, r_{K,k}) \in (\mathbb{F}_q^{Q\times(C+E)})^K \tag{6}$$

is a transpose of the k-th column of the result block. It reveals the validity of every transaction in the incoming strip $\mathbf{v}_k^{(t)}$. We will employ this notation in Section III-D.

Remark 3 (The degree of $F^{(t)}$)**.** The verification result of a transaction is the concatenation of functions *checkAddr* and *checkSig*. By definition, *checkAddr*$(\mathbf{p}, \mathbf{a}) =$ *hash1*$(\mathbf{p}) - \mathbf{a}$, where $\mathbf{a}$ is the output of function *fetch*$^{(t)}$. Besides, *checkSig*$(\mathbf{x}) = MQ(\mathbf{p}, \mathbf{s}) -$ *hash2*$(\mathbf{u}, \mathbf{p}, \mathbf{a})$, where the degree of the multivariate function $MQ(\mathbf{p}, \mathbf{s})$ is 3. Together, the degree of polynomial that verifies a transaction is $d = \max(T^{(t-1)} + 1, \deg \textit{hash1}, \deg \textit{hash2}, 3)$, where $T^{(t-1)} + 1$ is the degree of *fetch*$^{(t)}$.

As existing works [40, 41, 42] show the existence of secure polynomial hash functions with degree as low as 3, we assume that the degree of both *hash1* and *hash2* is less than $T^{(t-1)}$ in realistic blockchain systems (e.g., a blockchain system with 10^6 transactions in each shard has a verification function of degree $d = 20$). Hence, the polynomial $F^{(t)}$ has a degree $d = T^{(t-1)}+1$, which scales *logarithmically* with the number of transactions in a shard.

C. Coded Computation

Now that the verification of outgoing strips has been formulated as a low degree polynomial, we turn to describe how it is conducted in a coded fashion. In detail, every shard $k \in [K]$ is assigned a unique scalar $\omega_k \in \mathbb{F}_q$, and every node $i \in [N]$ is assigned a unique scalar $\alpha_i \in \mathbb{F}_q$.

Setting $T = 0$, the generator matrix in (1) becomes

$$G_{\mathcal{L}} = \begin{bmatrix} \Phi_1(\alpha_1) & \Phi_1(\alpha_2) & \ldots & \Phi_1(\alpha_N) \\ \vdots & \vdots & \ddots & \vdots \\ \Phi_K(\alpha_1) & \Phi_K(\alpha_2) & \ldots & \Phi_K(\alpha_N) \end{bmatrix}, \tag{7}$$

where $\Phi_k(z)$ is the Lagrange polynomial (2). For node i, the coded outgoing strip and coded incoming strip are linear combinations of outgoing strips and incoming strips, respectively, i.e.,

$$(\widetilde{\mathbf{h}}_i^{(t)})^\intercal = ((\mathbf{h}_1^{(t)})^\intercal, \ldots, (\mathbf{h}_K^{(t)})^\intercal) \cdot (G_{\mathcal{L}})_i = (\mathbf{B}^{(t)})^\intercal \cdot (G_{\mathcal{L}})_i,$$
$$(\widetilde{\mathbf{v}}_i^{(t)})^\intercal = ((\mathbf{v}_1^{(t)})^\intercal, \ldots, (\mathbf{v}_K^{(t)})^\intercal) \cdot (G_{\mathcal{L}})_i = \mathbf{B}^{(t)} \cdot (G_{\mathcal{L}})_i,$$

where $(G_{\mathcal{L}})_i$ is the i-th column of $G_{\mathcal{L}}$. Equivalently, $\widetilde{\mathbf{h}}_i^{(t)}$ and $\widetilde{\mathbf{v}}_i^{(t)}$ are evaluations of Lagrange polynomials $\psi^{(t)}(z)$ and $\phi^{(t)}(z)$ at α_i, respectively, where

$$\psi^{(t)}(z) = \sum_{k=1}^{K} \mathbf{h}_k^{(t)} \prod_{j\neq k} \frac{z-\omega_j}{\omega_k - \omega_j} \text{ and } \phi^{(t)}(z) = \sum_{k=1}^{K} \mathbf{v}_k^{(t)} \prod_{j\neq k} \frac{z-\omega_j}{\omega_k - \omega_j}.$$

Every node i stores a coded shard $\widetilde{\mathbf{V}}_i^{(t)}$, i.e., a node-specific linear combination of all shards,

$$\widetilde{\mathbf{V}}_i^{(t)} = \sum_{k=1}^{K} G_{k,i} \mathbf{V}_k^{(t)} = (\phi^{(1)}(\alpha_i), \ldots, \phi^{(t)}(\alpha_i)).$$

In epoch t, every node i receives the coded strips $\widetilde{\mathbf{h}}_i^{(t)}$ and $\widetilde{\mathbf{v}}_i^{(t)}$; protocols for secure encoding and delivery of coded strips are given in Section IV. Node i computes the polynomial verification function $F^{(t)}$ on $\widetilde{\mathbf{h}}_i^{(t)}$ and the locally stored $\widetilde{\mathbf{V}}_i^{t-1}$, and obtains a *coded outgoing result strip*

$$\widetilde{\mathbf{e}}_i^{(t)} = F^{(t)}(\widetilde{\mathbf{h}}_i^{(t)}, \widetilde{\mathbf{V}}_i^{(t-1)}) = F^{(t)}(\psi^{(t)}(\alpha_i), (\phi^{(1)}(\alpha_i), \ldots, \phi^{(t)}(\alpha_i)).$$

Formally, the coded outgoing result strip $\widetilde{\mathbf{e}}_i^{(t)}$, as well as the (uncoded) outgoing result strip $\mathbf{e}_k^{(t)}$ defined in Equation (4), is an evaluation of a polynomial $\mathbf{F}$, i.e.,

$$\mathbf{e}_k^{(t)} = \mathbf{F}^{(t)}(\omega_k) \text{ and } \widetilde{\mathbf{e}}_i^{(t)} = \mathbf{F}^{(t)}(\alpha_i), \tag{8}$$

$$\text{where } \mathbf{F}^{(t)}(z) = F^{(t)}(\psi^{(t)}(z), (\phi^{(1)}(z), \ldots, \phi^{(t)}(z)). \tag{9}$$

Since the degree of both $\psi^{(t)}$ and $\phi^{(t)}$ is $K-1$, it follows that the degree of $\mathbf{F}^{(t)}$ is $(K-1)d$.

Similar to the outgoing result strip defined in (4), the coded outgoing result strip

$$\widetilde{\mathbf{e}}_i^{(t)} = (\widetilde{\mathbf{e}}_{i,1}^{(t)}, \ldots, \widetilde{\mathbf{e}}_{i,K}^{(t)}) \in (\mathbb{F}_q^{Q\times(C+E)})^K \tag{10}$$

is a length-K vector, in which the k-th element $\widetilde{\mathbf{e}}_{i,k}^{(t)}$ contains Q entries and equals to the verification result of the k-th coded tiny block in the coded outgoing strip $\widetilde{\mathbf{h}}_i^{(t)}$. Note that unlike the coded incoming strip or the coded outgoing strip, the coded

outgoing result strip $\widetilde{\mathbf{e}}_i^{(t)}$ is *not* a linear combination of outgoing result strips $\mathbf{e}_1^{(t)}, \ldots, \mathbf{e}_K^{(t)}$ specified by $G_{\mathcal{L}}$. Instead, both of the coded $\widetilde{\mathbf{e}}_i^{(t)} = \mathbf{F}(\alpha_i)$ and uncoded $\mathbf{e}_i^{(t)} = \mathbf{F}(\omega_i)$ are evaluation of polynomial $\mathbf{F}(z)$ at different points (see Equation (12) and (13) for details).

Nodes further obtain the *indicator vector* $g \in \{0,1\}^{QK}$ by exchanging $\widetilde{\mathbf{e}}_i^{(t)}$; the details are given in Section IV-D. This data is crucial for the next section, namely *coded appending*, as each of its entries is associated with a coded transaction in every coded incoming strip. Specifically, note that every coded transaction is a linear combination of K transactions; the corresponding entry of the indicator vector g equals to 0 if they are all valid. Otherwise, if invalid transactions are included, the entry equals to 1.

D. Coded Appending

The appending operation of node i is instructed by the indicator vector g. Following the coded verification, each node i appends the coded *incoming* strip $\widetilde{\mathbf{v}}_i^{(t)}$ to their coded shard, after setting to zero the parts of it which failed the verification process. That is, node i zeros out the transactions whose corresponding entry of g equals to 1.

Remark 4. This process of setting to zero the parts which fail verification has an unexpected implication—it invalidates valid transactions that were linearly combined with invalid ones. We define the *Collateral Invalidation* (CI) rate as the number of transactions that are abandoned due to one invalid transaction, normalized by the total number of transactions processed in one epoch. Polyshard [29] has an CI rate of $\frac{1}{K}$, while our scheme has an CI rate of $\frac{1}{KQ}$, which is Q times smaller (i.e., better) than Polyshard.

IV. Coded Consensus

In this section, we discuss the consensus aspect of our design. Due to its coded nature, we propose three conditions that define *coded consensus*. Later, we show mechanisms that maintain these conditions. We use f to denote the number of Byzantine nodes, and define a *quorum* as a set of $N - f$ nodes. Our scheme tolerates these f Byzantine nodes in the partial synchrony model, given that $N \geq (K-1)d + 3f + 1$; a discussion on the nature of this assumption is given in the following section. Note that the communication between nodes is point-to-point, and the communication analysis takes into consideration every bit that is transmitted through the system.

First, our design must guarantee *consistency*, i.e, at every epoch t, correct nodes must perform coded verification on coded outgoing strips generated from the same block $\mathbf{B}^{(t)}$. Formally, we propose the following condition.

Condition 1 (Consistency). *Every correct node i obtains $\widetilde{\mathbf{h}}_i$, defined as $\widetilde{\mathbf{h}}_i^{(t)} = (G_{\mathcal{L}})_i^{\intercal} \cdot \mathbf{B}^{(t)}$, where $(G_{\mathcal{L}})_i$ denotes the i-th column of the generator matrix $G_{\mathcal{L}}$.*

This condition imposes that correct nodes obtain coded outgoing strips that are *consistent* with each other, i.e., correspond to the same block $\mathbf{B}^{(t)}$ defined in Equation (3). Otherwise, correct verification is impossible, as suggested in [11]. Moreover, our design must maintain *homology*.

Condition 2 (Homology). *Every correct node i obtains $\widetilde{\mathbf{v}}_i$, defined as $\widetilde{\mathbf{v}}_i = (G_{\mathcal{L}})_i^{\intercal}(\mathbf{B}^{(t)})^{\intercal}$, where both $\mathbf{B}^{(t)}$ and $G_{\mathcal{L}}$ are as in Condition 1.*

The second condition suggests that every node obtains the coded incoming strip that is *homologous* to the coded outgoing strip, i.e., generated from the same block $\mathbf{B}^{(t)}$. Otherwise, we say they are *nonhomologous*; such nonhomology problem can cause a discrepancy between the verified and the appended, i.e., nodes verify valid transactions, but append invalid ones, nullifying the verification efforts. Satisfying this condition assures the correct appending of incoming strips. Finally, the blockchain must not store invalid transactions, which gives rise to the last condition.

Condition 3 (Validity). *Every correct node i appends the coded incoming strip $\widetilde{\mathbf{v}}_i^{(t)}$ to its local coded chain after setting the invalid coded transactions to zero, i.e, coded transactions which were not formed exclusively from valid transactions.*

This condition requires every node i to obtain the indicator vector g defined in Section III-D. Together, we say that a protocol provides *coded consensus* if it simultaneously achieves Condition 1, Condition 2 and Condition 3, i.e., maintains consistency, homology, and validity at the same time. We propose such a protocol, employing a leader to distribute coded strips and provide coded consensus. Our approach adopts HotStuff [20], a BFT SMR with linear message complexity, and techniques from Information Dispersal for consistency and homology. In addition, we employ coded computation that maintains validity of the system. Further, the superscript (t) is omitted for clarity in the rest of this paper.

A. Overview

In order to maintain the aforementioned three properties, we employ HotStuff to maintain a chain of *headers*, each corresponds to a block. HotStuff provides the safety and liveness property of the header chain. Together with information dispersal techniques, our scheme maintains the consistency property. We provide detailed discussion in Section IV-B. Further,

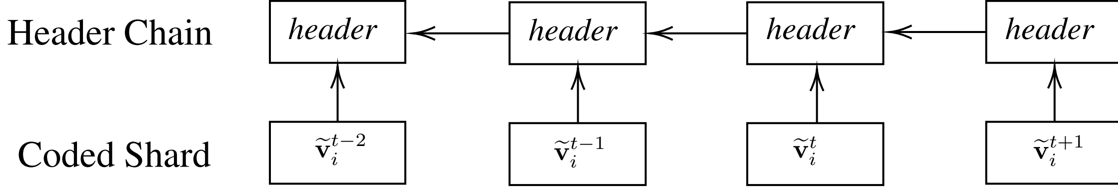

Fig. 2. Illustration of the internal storage of node i. We use HotStuff for nodes to reach a consensus on a chain of headers. Meanwhile, each node stores a distinct chain of coded incoming strips. i.e., the coded shard. The consensus on the chain of headers assures that at any height t (i.e., epoch) of the chain, nodes store coded outgoing strips that are consistent with each other, i.e., generated from the same block $\mathbf{B}$

we incorporate extra mechanisms in HotStuff to maintain homology (Section IV-C) and validity (Section IV-D). In Section V, we show that our scheme indeed provides coded consensus, and inherits the liveness property from HotStuff.

B. Maintaining Consistency (Condition 1)

We first address the consistency problem of coded outgoing strips generated from $\mathbf{B}$ (the case for coded incoming strips are similar). Our method depends on a data structure called *checksum*, introduced by AVID-FP [32]. A checksum allows nodes to verify that the received coded strip is consistent with ones received by others, i.e., computed from the same block. It contains a list of K *fingerprints*. Each fingerprint is generated from an (uncoded) outgoing strip (a row of $\mathbf{B}$), using some ε-fingerprinting function *fp* defined as follows.

Definition 1. [32, Definition 2.1] A function $fp : T \times \mathbb{F}_q^\delta \to \mathbb{F}_q^\gamma$ is an ε-fingerprinting function if

$$\max_{d,d' \in \mathbb{F}_q^\delta, d \neq d'} \Pr_{r \sim Unif(T)} [fp(r,d) = fp(r,d')] \leq \varepsilon.$$

That is, the probability for two distinct $d, d' \in \mathbb{F}_q^\delta$ to have the same fingerprint is at most ε, where the key r is chosen uniformly at random from some input space T.

Examples of ε-fingerprinting functions include *division fingerprinting*, which generalizes Rabin's fingerprinting [44] from $\mathbb{F}_2$ to any field $\mathbb{F}_q$. With coefficients in $\mathbb{F}_q$, the input $d \in \mathbb{F}_q^\delta$ is regarded as a polynomial $d(x)$ of degree δ, and T is a collection of monic irreducible polynomials of degree γ. The division fingerprinting function returns the remainder of $d(x)$ divided by $p(x)$, i.e., $d(x)$ mod $p(x)$, where $p(x)$ is chosen from T uniformly at random.

We let $fp : T \times \mathbb{F}_q^\delta \to \mathbb{F}_q^\gamma$ be an ε-fingerprinting function where $\delta = \frac{|\mathbf{B}|}{K}$ is the size of a strip. As done in AVID-FP [32], the random selection of r from T is simulated by deterministic cryptographic hash functions [45]; referring to the use of a hash function as a random oracle is a common practice in blockchain systems, e.g., in [24]. In addition to the K fingerprints, a list of hash values $cc = [hash(\widetilde{\mathbf{h}}_1), \ldots, hash(\widetilde{\mathbf{h}}_N)]$ is included in the checksum, generated using a cryptographic hash function $hash : \mathbb{F}_q^* \to \mathbb{F}_q^\lambda$ (not to be confused with *hash1* and *hash2* mentioned earlier). The selection of r is achieved by another cryptographic hash function $select : (\mathbb{F}_q^\lambda)^N \to T$, which takes the list cc as input and outputs an element in T[2]. Formally, the function CHECKSUM in Algorithm 1 encapsulates the construction of the checksum.

The leader first generates coded outgoing strips $(G_\mathcal{L})_1^\intercal \cdot \mathbf{B}, \ldots, (G_\mathcal{L})_N^\intercal \cdot \mathbf{B}$ and constructs the checksum *cksH*. Similarly, it creates coded incoming strips $(G_\mathcal{L})_1^\intercal \cdot \mathbf{B}^\intercal, \ldots, (G_\mathcal{L})_N^\intercal \cdot \mathbf{B}^\intercal$ and *cksV*. The leader then sends the checksums to every node i piggybacked with the coded fragments $\widetilde{\mathbf{h}}_i = (G_\mathcal{L})_i^\intercal \cdot \mathbf{B}$ and $\widetilde{\mathbf{v}}_i = (G_\mathcal{L})_i^\intercal \cdot \mathbf{B}^\intercal$. In order to verify that a coded strip *agrees* with the received checksum, i.e., *cksH* and $\widetilde{\mathbf{h}}_i$ are computed from the same $\mathbf{B}$, and *cksV* and $\widetilde{\mathbf{v}}_i$ are computed from the same $\mathbf{B}^\intercal$, we require the fingerprinting function to be homomorphic.

Definition 2. [32, Definition 2.5] A fingerprinting function $fp : T \times \mathbb{F}_q^\delta \to \mathbb{F}_q^\gamma$ is *homomorphic* if $fp(r,d) + fp(r,d') = fp(r, d+d')$ and $b \cdot fp(r,d) = fp(r, b \cdot d)$ for any $r \in T$, any $b \in \mathbb{F}_q$, and any $d, d' \in \mathbb{F}_q^\delta$.

This property enables the node to verify that the coded fragment satisfies the required linear combination (defined by the generator matrix) with uncoded strips, by having access only to the fingerprints of uncoded strips, and not to the uncoded strips. As any coded strip is a linear combination of K uncoded strips, the homomorphism guarantees that its fingerprint must be equal to the same linear combination of K fingerprints of uncoded strips, i.e.,

$$(G_\mathcal{L})^\intercal \begin{bmatrix} fp(r, \mathbf{h}_1) \\ \vdots \\ fp(r, \mathbf{h}_K) \end{bmatrix} = \begin{bmatrix} fp(r, \widetilde{\mathbf{h}}_1) \\ \vdots \\ fp(r, \widetilde{\mathbf{h}}_N) \end{bmatrix}, \text{ and } (G_\mathcal{L})^\intercal \begin{bmatrix} fp(r, \mathbf{v}_1) \\ \vdots \\ fp(r, \mathbf{v}_K) \end{bmatrix} = \begin{bmatrix} fp(r, \widetilde{\mathbf{v}}_1) \\ \vdots \\ fp(r, \widetilde{\mathbf{v}}_N) \end{bmatrix}.$$

[2] We point out that Verifiable Random Function (VRF) [33] is an alternative implementation of the random oracle, which is communication-efficient as it does not require the checksum to contain N hash values. It has been employed in blockchain designs including [9] and [25], but for different purposes.

Note that $\mathbf{h}_k$ is the k-th row of the matrix $\mathbf{B}$, and $\widetilde{\mathbf{h}}_i$ is the i-th row of the matrix $\widetilde{\mathbf{B}} = (G_{\mathcal{L}})_i^{\intercal}\mathbf{B}$. Similarly, $\mathbf{v}_k$ is the k-th row of the matrix $\mathbf{B}^{\intercal}$, and $\widetilde{\mathbf{v}}_i$ is the i-th row of the matrix $\widetilde{\mathbf{B}^{\intercal}} = (G_{\mathcal{L}})_i^{\intercal}\mathbf{B}^{\intercal}$. Thus, each node can confirm the agreement between the received (coded) strip and the checksum, as long as the fingerprints of the coded strip match the encoding of the K fingerprints in the checksum; this is guaranteed with high probability, as shown [32, Theorem 3.4].

In this respect, each node can assure that the received coded strip is consistent with ones received by others by *reaching a consensus on the checksum* which is in agreement with all coded strips. Treating checksums as requests, we can employ BFT SMR protocols that allow correct nodes to reach a consensus on the total order of them, and hence maintain the consistency of strips at any epoch t. Specifically, we adopt HotStuff [20], a leader-based BFT SMR protocol that works in partial synchrony (see Section II-B). We define a *header* of a block $\mathbf{B}$ as a concatenation of checksums computed from the matrix $\mathbf{B}$ and its transpose $\mathbf{B}^{\intercal}$, i.e.,

$$header = (cksH, cksV).$$

HotStuff allows nodes to reach a consensus on a chain of header. HotStuff always ensure *safety* given bounded number of faulty nodes ($N \geq 3f + 1$). That is, no two correct node should accept conflicting headers; by conflicting we mean the chain led by neither one extends the chain led by the other. Hence, correct nodes will never accept different headers at any epoch t. When the system becomes synchronous, HotStuff provides the *liveness* property, such that the consensus on headers will be reached when the leader is correct. As discussed earlier, such a consensus on a header guarantees the consistency of coded fragment generated from the corresponding block.

For clarity, the lines in Algorithm 2 and Algorithm 3 are color coded. The pseudocode of HotStuff is provided in *black*. The blue lines concern the distribution and verification of coded strips, maintaining validity. The orange lines and green lines maintain consistency and homology, respectively. We will elaborate the colored lines in sequel. In particular, we argue that our add-ons do not affect the safety and liveness property of HotStuff algorithm.

Algorithm 1 Utilities

```
1:  function MSG(type, header, qc, payload)
2:    m.type = type
3:    m.viewNumber = curView
4:    m.header = header
5:    m.qc = qc
6:    m.payload = payload
7:    return m
8:  function HEADER(prev, checksums)
9:    header.prev = prev
10:   header.checksums = checksums
    // instantiate QC from a set of messages
11: function QC(M)
12:   qc.type ← m.type : m ∈ M
13:   qc.viewNumber ← m.viewNumber : m ∈ M
14:   qc.header ← m.header : m ∈ M
15:   qc.sig ← tcombine(⟨qc.type, qc.viewNumber,
                  {qc.header⟩, m.partialSig | m ∈ M})
16: function MATCHINGMSG(m, t, v)
17:   return (m.type = t) ∧ (m.viewNumber = v)
18: function MATCHINGQC(qc, t, v)
19:   return (qc.type = t) ∧ (qc.viewNumber = v)∧
      tverify(⟨qc.type, qc.viewNumber, qc.header⟩, qc.sig)
20: function ENCODE(G, m)                     // m:length-K vector
21:   return m · G
22: function ENCODEROW(G, M)
23:   return G^T · M
24: function CHECKSUM(G, M)
25:   M~ ← ENCODEROW(G, M)                    // M~: N × K matrix
26:   for k = 1 to N do
27:     cks.CC[i] ← hash(m~_{i,*})            // the i-th row of M~
28:   r ← select(cks.CC)
29:   for k = 1 to K do
30:     cks.FP[k] ← fp(r, m_{k,*})            // the k-th row of M
31:   return cks, M~
    // check if checksum agrees with coded fragment
32: function AGREE(checksum, fragment, i)
33:   h ← hash(fragment) f ← fp(select(cks.CC), fragment)
34:   return (h = cks.CC[i]) ∧ (f = ENCODE(G_L, cks.FP)[i])
35: function SAFEHEADER(header, qc)
36:   return (header extends from lockedQC.header)∨
37:          (qc.viewNumber > lockedQC.viewNumber)
38: function SIGNEACH(m, i)                   // m: length-N vector
39:   for j = 1 to N do result[j] ← ⟨m[j]⟩_{σ_i}
40:   return result
```

HotStuff runs in consecutive *views* associated with increasing integer view numbers. In each view there is a designated node LEADER(*viewNumber*) that proposes new headers and distribute coded strips. In order to append a header to the chain, the leader must collect partial signatures on its proposal from a quorum of nodes in each of three phases, namely PREPARE, PRE-COMMIT and COMMIT. The partial signatures are generated with a $(N - f, N)$-threshold signature scheme π.

A new leader must collect NEW-VIEW messages from a quorum of nodes; a correct node sends the NEW-VIEW message, alongside a valid *prepareQC* (define next) with highest view that it has received, to the leader of the next view if it believes the current one fails (line 22, Algorithm 3). The new leader chooses the one, called *highQC*, with the highest view number within all received *prepareQC*s. It creates and extends the new header (i.e., containing the hash value of another header) from the header contained in *highQC*. If the leader is an incumbent one, it extends the header from its last proposed header.

Incumbent or not, the leader sends the newly created header to every node i piggybacked with the corresponding coded strips $\widetilde{\mathbf{h}}_i$ and $\widetilde{\mathbf{v}}_i$ (line 8, Algorithm 2) generated from the block $\mathbf{B}$ (line 2, Algorithm 2). Upon receiving the PREPARE message from the leader, the node i runs the function SAFEHEADER in Algorithm 1 which compares the newly received header with the header it has locked on (i.e. the header contained in the *precommitQC* with highest view number it has received, called *lockedQC*, which will be defined next). The new header is considered valid if it extends from the locked header, or extends from a header

Algorithm 2 Coded Consensus Part 1

▷ PREPARE PHASE
1: **as** a leader // $i = \text{LEADER}(curView)$
2: $\mathbf{B} \leftarrow$ collect a block of transactions
3: $(cksH, \widetilde{\mathbf{B}}) \leftarrow \text{CHECKSUM}(G_{\mathcal{L}}, \mathbf{B}), (cksV, \widetilde{\mathbf{B}^{\intercal}}) \leftarrow \text{CHECKSUM}(G_{\mathcal{L}}, \mathbf{B}^{\intercal})$
4: **wait for** $N - f$ NEW-VIEW messages: $\mathcal{M} \leftarrow \{m \mid \text{MATCHINGMSG}(m, \text{NEW-VIEW}, curView - 1)\}$
5: $highQC \leftarrow (\arg\max_{m \in \mathcal{M}} \{m.qc.viewNumber\}).qc$ // QC with highest view number
6: $header \leftarrow \text{HEADER}(highQC.header, [cksH, cksV])$
7: **for** $i = 1$ to N **do**
8: $payload \leftarrow [\widetilde{\mathbf{h}}_i, \widetilde{\mathbf{v}}_i]$ // the i-th rows of $\widetilde{\mathbf{B}}$ and $\widetilde{\mathbf{B}^{\intercal}}$
9: send $\text{MSG}(\text{PREPARE}, header, highQC, payload)$ to node i
10: **as** node i
11: **wait for** message m from $\text{LEADER}(curView)$: $m \leftarrow \text{MATCHINGMSG}(m, \text{PREPARE}, curView)$
12: **if** $(m.header$ extends from $m.qc.header) \wedge (\text{SAFEHEADER}(m.header, header.qc))$ **then**
13: **if** $\neg(\text{AGREE}(header.checksum[1], m.payload[1], i) \wedge \text{AGREE}(header.checksum[2], m.payload[2], i)$ **then**
14: $codedResults \leftarrow F(\widetilde{\mathbf{V}}_i, \widetilde{\mathbf{h}}_i)$
15: $w_{i,*} \leftarrow \text{ENCODE}(G_{\mathcal{L}}, m.payload[1])$, $px_i \leftarrow \text{SIGNEACH}(w_{i,*})$
16: $u_{i,*} \leftarrow \text{ENCODE}(G_{\mathcal{L}}, m.payload[2])$
17: $results \leftarrow \text{ENCODE}(G_{\mathcal{L}}, codedResults)$, $sigResults \leftarrow \text{SIGNEACH}(\text{ENCODE}(G_{\mathcal{L}}, codedResults))$
18: $ack \leftarrow \text{MSG}(\text{PREPARE}, m.header, \perp, payload \leftarrow (px_i, results, sigResults)$
19: $ack.partialSig \leftarrow \langle \text{PREPARE}, curView, m.header \rangle_{\pi,i}$
20: send ack to LEADER(curView) // acknowledge the PREPARE message

▷ PRE-COMMIT PHASE
21: **as** a leader
22: **wait for** ack's on PREPARE from a quorum $\mathcal{I}$: $\mathcal{A} \leftarrow \{ack \mid \text{MATCHINGMSG}(ack, \text{PREPARE}, curView)\}$
23: $prepareQC \leftarrow QC(\mathcal{A})$
24: **for** $i = 1$ to N **do**
25: $payload[1] \leftarrow \text{COLUMN}(\{ack.payload[1] \mid ack \in \mathcal{A}\}, i)$
26: $payload[3] \leftarrow \text{COLUMN}(\{ack.payload[3] \mid ack \in \mathcal{A}\}, i)$
27: $payload[4] \leftarrow \text{COLUMN}(\{ack.payload[4] \mid ack \in \mathcal{A}\}, i)$
28: $m \leftarrow \text{MSG}(\text{PRE-COMMIT}, \perp, prepareQC, payload)$, $m.quorumIdentifier \leftarrow \text{QI}(\mathcal{I})$
29: send m to node i
30: **as** node i
31: **wait for** message m from $\text{LEADER}(curView)$: $m \leftarrow \text{MATCHINGQC}(m.qc, \text{PRE-COMMIT}, curView)$
32: $prepareQC \leftarrow m.qc$
33: **if** $\text{VERIFYSIG}(m.payload[1]), u_{i,*}) \wedge \text{VERIFYSIG}(m.payload[2], m.payload[3])$ **then**
34: $decoded \leftarrow \text{DECODE}(m.payload[3])$
35: $binaryResults \leftarrow \text{BINARY}(decoded)$
36: $payload \leftarrow \text{PARTIALINDICATOR}(binaryResults)$
37: $ack \leftarrow \text{MSG}(\text{PRE-COMMIT}, m.qc.header, \perp, payload)$
38: $ack.partialSig \leftarrow \langle \text{PRE-COMMIT}, curView, m.qc.header \rangle_{\pi,i}$, $ack.partialSigQI \leftarrow \langle QI \rangle_{\pi,i}$
39: send ack to LEADER(curView) // acknowledge the PRE-COMMIT message

in a *prepareQC* with a higher view number than the *lockedQC* (line 37, Algorithm 1). Such a check guarantees both safety [20, Theorem 2] and liveness [20, Theorem 4].

Further, a valid checksum in the header must agree with the coded strip, and we implement the checking process in the function AGREE in Algorithm 1. In particular, we add another predicate in line 13 of Algorithm 2 to verify such agreements.

If the aforementioned two predicates both return true, node i responds with a partial signature $\langle \text{PREPARE}, curView, header \rangle_{\pi,i}$ acknowledging the header from the leader of the current view (line 19, Algorithm 2). The leader then enters the PRE-COMMIT phase, and instantiates a *prepareQC* (where *QC* stands for *quorum certificate*) from the replies using the constructor function QC in Algorithm 1. The *prepareQC* contains a valid signature $\langle \text{PREPARE}, curView, header \rangle_{\pi}$, showing a quorum of $N - f$ nodes has acknowledged the PREPARE message from the current leader.

The leader broadcasts *prepareQC*. Every node verifies the signature $\langle \text{PREPARE}, curView, header \rangle_{\pi}$ using the function MATCHINGQC in Algortithm 1. After that, node i replies with a partial signature $\langle \text{PRE-COMMIT}, curView, header \rangle_{\pi,i}$. From the replies the leader creates a *precommitQC*, and broadcast it in the COMMIT phase. Similarly, from the replies the leader creates *commitQC*; nodes only link the coded outgoing strip $\widetilde{\mathbf{v}}_i$ to the local chain after receiving the *commitQC*.

C. Maintaining Homology (Condition 2)

Although [20] allows nodes to reach a consensus on the chain of headers, which guarantees the consistency property of strips, the homology problem remains. With a Byzantine leader, even though a correct node i may obtain a consistent coded outgoing strip $\widetilde{\mathbf{h}}_i = (G_{\mathcal{L}})_i^{\intercal}\mathbf{B}$ and consistent coded incoming strip $\widetilde{\mathbf{v}}_i = (G_{\mathcal{L}})_i^{\intercal}(\mathbf{B}')^{\intercal}$, they might correspond to different blocks $\mathbf{B} \neq \mathbf{B}'$. To solve this problem, we integrate the following design (in green) with Hotstuff's three-phase protocol to maintain homology.

Algorithm 3 Coded Consensus Part 2

▷ COMMIT PHASE
1: **as** a leader
2: **wait for** $(N-f)$ *ack*'s on PRE-COMMIT: $\mathcal{A} \leftarrow \{ack \mid$ MATCHINGMSG(*ack*, PRE-COMMIT, *curView*)$\}$
3: *payload* ← MERGEINDICATORS($\mathcal{A}$)
4: *precommitQC* ← $QC(\mathcal{A})$
5: m ← MSG(COMMIT, $\perp$, *precommitQC*, *payload*)
6: *m.signatureQI* ← *tcombine*($\langle QI \rangle$, $\{$*ack.partialSigQI* $\mid ack \in \mathcal{A}\}$)
7: broadcast m
8: **as** node i
9: **wait for** message m from LEADER(*curView*): m ← MATCHINGQC(m, COMMIT, *curView*)
10: **if** *tverify*($\langle QI(\mathcal{I})\rangle$, *m.signatureQI*) **then**
11: *lockedQC* ← *m.qc*
12: $\widetilde{\mathbf{v}}_i$ ← UPDATE($\widetilde{\mathbf{v}}_i$, *m.payload*) // update coded incoming strip using g
13: *ack* ← MSG(COMMIT, *m.qc.header*, $\perp, \perp$), *ack.partialSig* ← $\langle$COMMIT, *curView*, *m.qc.header*$\rangle_{\pi,i}$
14: send *ack* to LEADER(curView) // acknowledge the PRE-COMMIT message

▷ DECIDE PHASE
15: **as** a leader
16: **wait for** $(N-f)$ *ack*'s on COMMIT: $\mathcal{A} \leftarrow \{ack \mid$ MATCHINGMSG(*ack*, COMMIT, *curView*)$\}$
17: *commitQC* ← $QC(\mathcal{A})$
18: broadcast MSG(DECIDE, $\perp$, *commitQC*, $\perp$)
19: **as** node i
20: **wait for** message m from LEADER(*curView*): m ← MATCHINGQC(m, COMMIT, *curView*)
21: append $\widetilde{\mathbf{v}}_i$ to local chain

▷ NEXTVIEW INTERRUPT
22: send MSG(NEW-VIEW, $\perp$, *prepareQC*, $\perp$) to LEADER(*curView* + 1)

Upon receiving the coded outgoing strip $\widetilde{\mathbf{h}}_i = (G_{\mathcal{L}})_i^\intercal \mathbf{B}$ from the leader in the PREPARE phase, node i multiplies it from the right with $G_{\mathcal{L}}$, creating a length-N vector $w_{i,*}$, which equals to the i-th row of the matrix $\mathbf{W} = G_{\mathcal{L}}^\intercal \mathbf{B} G_{\mathcal{L}}$ (line 15, Algorithm 2). Similarly, it creates a vector $u_{i,*} = (G_{\mathcal{L}})_i^\intercal \widetilde{\mathbf{v}}_i$ as the i-th row of $\mathbf{U} = (G_{\mathcal{L}})^\intercal (\mathbf{B}')^\intercal G_{\mathcal{L}}$ (line 16, Algorithm 2).

Node i then defines a length-N *signature vector* px_i, whose j-th entry stores its digital signature (not to be confused with partial signature) on $\langle w_{i,j}, j\rangle$. Formally, we have

$$px_i[j] = \langle w_{i,j}, j\rangle_{\sigma_i}, \text{ for } j \in [N]. \tag{11}$$

Node i sends back px_i in the acknowledgement of the PREPARE message received from the leader. For every received px_i, the leader first verifies if $px_i[j]$ is indeed a valid signature on $w_{i,j}$, for each $j \in [N]$. This step is omitted in the pseudocode for clarity, and the leader ignores messages that fail the verification. After collecting such vectors from nodes in a quorum $\mathcal{I}$ of size $|\mathcal{I}| = N - f$, the leader stacks the px_i's in an $(N-f)\times N$ matrix ordered by the indices of nodes. It sends the j-th column of the resulting matrix to every node $j \in [N]$ in the PRE-COMMIT message together with a *quorum identifier* $QI(\mathcal{I})$ that specifies the members of $\mathcal{I}$.[3] (line 28, Algorithm 2).

Upon receiving the PRE-COMMIT message from the leader, node j learns the members of $\mathcal{I}$ from the quorum identifier $QI(\mathcal{I})$. Meanwhile, node j receives $\langle w_{i,j}\rangle_{\sigma_i}$, for every $i \in \mathcal{I}$, and verifies if the received $\langle w_{i,j}, j\rangle_{\sigma_i}$ is a valid signature on $\langle u_{j,i}, j\rangle$ (line 33, Algorithm 2). The process is encapsulated in function VERIFYSIG, whose simple implementation (see above) is omitted for brevity. If the verification passes, node j creates partial signature $\langle QI(\mathcal{I})\rangle_{\pi,i}$ on the quorum identifier and sends it back to the leader as an acknowledgement of the PRE-COMMIT message (line 38, Algorithm 2).

The leader verifies the received partial signature; this verification is omitted in the pseudocode. Upon receiving acknowledgements from a quorum $\mathcal{J}$ of nodes, the leader combines partial signatures and broadcasts a COMMIT message with a valid signature $\langle QI(\mathcal{I})\rangle_\pi$ (line 6–7, Algorithm 3). Nodes can be convinced that $\mathbf{B} = \mathbf{B}'$ after verifying $\langle QI(\mathcal{I})\rangle_\pi$ in the COMMIT message, due to the following lemma.

Lemma 1. *A valid signature* $\langle QI(\mathcal{I})\rangle_\pi$ *implies* $\mathbf{B} = \mathbf{B}'$.

Proof. The signature $\langle QI(\mathcal{I})\rangle_\pi$ reveals the existence of a quorum $\mathcal{J}$ such that for every correct node $j \in \mathcal{J}$ and every correct node $i \in \mathcal{I}$, we have

$$w_{i,j} = (G_{\mathcal{L}})_i^\intercal \mathbf{B} (G_{\mathcal{L}})_j = (G_{\mathcal{L}})_j^\intercal (\mathbf{B}')^\intercal (G_{\mathcal{L}})_i = u_{j,i}.$$

Since we assume that $N = (K-1)d + 3f + 1$, the quorums $\mathcal{I}$ and $\mathcal{J}$ intersect on at least

$$2(N-f) - N = (K-1)d + f + 1 - (K-1) + (K-1) = (K-1)(d-1) + f + K \geq f + K$$

nodes, which contains at least K correct ones.

[3]Since there are $\binom{N}{f}$ possible quorums, $\log\binom{N}{f} < \log[\sum_{f=1}^{N}\binom{N}{f}] = \log 2^N = N$ bits suffice to uniquely present either of them; this is negligible in size compared to the $N - 2f$ digital signatures sent along with it.

Let $\mathcal{K}$ be a set containing these K correct nodes, and let $G_{\mathcal{K}}$ be a $K \times K$ matrix containing the corresponding K columns of the Lagrange matrix $G_{\mathcal{L}}$. Since $g_k^\intercal \mathbf{B} g_{k'} = g_{k'}^\intercal (\mathbf{B}')^\intercal g_k$ for every $k, k' \in [K]$, it follows that $G_{\mathcal{K}}^\intercal \mathbf{B} G_{\mathcal{K}} = G_{\mathcal{K}}^\intercal \mathbf{B}' G_{\mathcal{K}}$. By the MDS property of $G_{\mathcal{L}}$, the matrix $G_{\mathcal{K}}$ is invertible, and hence $\mathbf{B} = \mathbf{B}'$. □

D. Maintaining Validity (Condition 3)

So far, we have developed mechanisms that maintain homology and consistency. Together, every correct node i is performing verification on the coded outgoing strip $\widetilde{\mathbf{h}}_i = (G_{\mathcal{L}})_i^\intercal \mathbf{B}$ and appending the coded incoming strip $\widetilde{\mathbf{v}}_i = (G_{\mathcal{L}})_i^\intercal \mathbf{B}^\intercal$. We now present a communication-efficient scheme that employs coded computation to guarantee validity, such that no invalid transactions in the block can be appended to the blockchain.

Specifically, we weave a mechanism into the existing protocol. It allows nodes to securely obtain the indicator vector $g \in \{0,1\}^{QK}$. Note that each of the entries of g is associated with a coded transaction in every coded incoming strip. A coded transaction should be zeroed-out if the corresponding entry is 1 (see Section III-C).

Recall that the degree of the polynomial verification function $\mathbf{F}(z)$ is $(K-1)d$, and hence it is uniquely defined by evaluations at any $L = (K-1)d + 1$ distinct points. That is, for any distinct $\beta_1, \ldots, \beta_L$, one may represent $\mathbf{F}(z)$ as a linear combination of L Lagrange basis polynomials $\Psi_1(z), \ldots, \Psi_L(z)$, i.e.,

$$\mathbf{F}(z) = \sum_{\ell \in [L]} \mathbf{F}(\beta_i) \Psi_\ell(z), \text{ where } \Psi_\ell(z) = \prod_{l, \ell \in [L], l \neq \ell} \frac{z - \beta_l}{\beta_\ell - \beta_l}.$$

As a result, the coded outgoing result strips $\mathbf{F}(\alpha_1), \ldots, \mathbf{F}(\alpha_N)$ can be represented as

$$\begin{bmatrix} \widetilde{\mathbf{e}}_1 \\ \vdots \\ \widetilde{\mathbf{e}}_N \end{bmatrix} \overset{(8)}{=} \begin{bmatrix} \mathbf{F}(\alpha_1) \\ \vdots \\ \mathbf{F}(\alpha_N) \end{bmatrix} = G_{\mathcal{F},\alpha}^\intercal \cdot \begin{bmatrix} \mathbf{F}(\beta_1) \\ \vdots \\ \mathbf{F}(\beta_L) \end{bmatrix} = \begin{bmatrix} \Psi_1(\alpha_1) & \Psi_1(\alpha_2) & \ldots & \Psi_1(\alpha_N) \\ \Psi_2(\alpha_1) & \Psi_2(\alpha_2) & \ldots & \Psi_2(\alpha_N) \\ \vdots & \vdots & \ddots & \vdots \\ \Psi_L(\alpha_1) & \Psi_L(\alpha_2) & \ldots & \Psi_L(\alpha_N) \end{bmatrix}^\intercal \cdot \begin{bmatrix} \mathbf{F}(\beta_1) \\ \vdots \\ \mathbf{F}(\beta_L) \end{bmatrix}, \tag{12}$$

and the (uncoded) outgoing result strips can be represented as

$$\begin{bmatrix} \mathbf{e}_1 \\ \vdots \\ \mathbf{e}_K \end{bmatrix} \overset{(8)}{=} \begin{bmatrix} \mathbf{F}(\omega_1) \\ \vdots \\ \mathbf{F}(\omega_K) \end{bmatrix} = G_{\mathcal{F},\omega}^\intercal \cdot \begin{bmatrix} \mathbf{F}(\beta_1) \\ \vdots \\ \mathbf{F}(\beta_L) \end{bmatrix} = \begin{bmatrix} \Psi_1(\omega_1) & \Psi_1(\omega_2) & \ldots & \Psi_1(\omega_K) \\ \Psi_2(\omega_1) & \Psi_2(\omega_2) & \ldots & \Psi_2(\omega_K) \\ \vdots & \vdots & \ddots & \vdots \\ \Psi_L(\omega_1) & \Psi_L(\omega_2) & \ldots & \Psi_L(\omega_K) \end{bmatrix}^\intercal \cdot \begin{bmatrix} \mathbf{F}(\beta_1) \\ \vdots \\ \mathbf{F}(\beta_L) \end{bmatrix}. \tag{13}$$

Upon receiving the message from the leader in the PREPARE phase, node i computes the verification function $\mathbf{F}$ and obtains its coded outgoing result strip $\widetilde{\mathbf{e}}_i$ (line 14, Algorithm 2). Node i multiplies it from the right with the Lagrange matrix $G_{\mathcal{L}} \in \mathbb{F}_q^{K \times N}$, and obtains $c_{i,*} = (\widetilde{\mathbf{e}}_{i,1}, \ldots, \widetilde{\mathbf{e}}_{i,K}) \cdot G_{\mathcal{L}}$, which equals to the i-th row of the matrix

$$\mathbf{C} = G_{\mathcal{F},\alpha}^\intercal \cdot \left[\mathbf{F}(\beta_1)^\intercal, \ldots, \mathbf{F}(\beta_L)^\intercal\right]^\intercal \cdot G_{\mathcal{L}}. \tag{14}$$

In the acknowledgment of the PREPARE message, node i replies the leader with $c_{i,*}$ with its signatures on each entry (line 17, Algorithm 2). The leader verifies if the signatures matches $c_{i,*}$; this step is omitted in the pseudocode for clarity. Upon receiving a quorum of $N - f$ such vectors, the leader stacks them on top of each other to form a $(N-f) \times N$ matrix (which is a submatrix of $\mathbf{C}$), and sends the j-th column to every node $j \in [N]$ in the PRE-COMMIT phase (line 26, Algorithm 2). Note that the j-th column of $\mathbf{C}$ is the encoding of the j-th column of matrix $[\mathbf{F}(\beta_1)^\intercal, \ldots, \mathbf{F}(\beta_L)^\intercal]^\intercal \cdot G_{\mathcal{L}}$ using the generator matrix $G_{\mathcal{F},\alpha}$, which generates a Lagrange code of length N and dimension L. As a result, every node can perform Reed-Solomon decoding after verifying the signature of each entry (line 33, Algorithm 2), and obtain the j-th column of matrix $[\mathbf{F}(\beta_1)^\intercal, \ldots, \mathbf{F}(\beta_L)^\intercal]^\intercal \cdot G_{\mathcal{L}}$ (line 34, Algorithm 2). The decoding is given in the function DECODE which calls a Reed-Solomon decoder. Since we have $N \geq (K-1)d + 3f + 1$, decoding from $N - f$ elements will be successful since there are at most f Byzantine nodes.

By left multiplying the decoded column with $G_{\mathcal{F},\omega}^\intercal$, every correct node i obtains the j-th column of the matrix $G_{\mathcal{F},\omega}^\intercal \cdot \left[\mathbf{F}(\beta_1)^\intercal, \ldots, \mathbf{F}(\beta_L)^\intercal\right]^\intercal \cdot G_{\mathcal{L}}$. This vector equals to the j-th column of $\left[\mathbf{e}_1^\intercal, \ldots, \mathbf{e}_K^\intercal\right]^\intercal \cdot G_{\mathcal{L}}$ by Equation (13), which further equals to the j-th coded incoming result strip by Equation (5) and Equation (6), i.e., $\widetilde{\mathbf{s}}_j = \mathbf{R} \cdot (G_{\mathcal{L}})_j$.

Recall that the result block $\mathbf{R}$ is a $K \times K$ matrix whose each element $r_{k,k'}$ stores the verification results of the Q transactions in the tiny block $b_{k,k'}$. Therefore, a coded incoming results strip $\widetilde{\mathbf{s}}_i$ contains K coded tiny result blocks; the k-th one is a linear combination of $r_{k,1}, \ldots, r_{k,K}$ defined by $(G_{\mathcal{L}})_k$. Hence, for $l \in [Q]$, the l-th entry in the k-th coded tiny result block is a linear combination of verification results of transactions in $\mathcal{S}_{k,l}$; a set containing every l-th transaction in $b_{k,1}, \ldots, b_{k,K}$. If the entry is not a zero vector, it suggests that at least one of these verification results is not a zero vector, which further suggests at least one transaction in $\mathcal{S}_{k,l}$ is invalid. On the other hand, if the entry is a zero vector, node i *cannot* conclude the validity of transactions in $\mathcal{S}_{k,l}$, as a linear combination of non-zero vectors might be the zero vector.

Recall that in an MDS code of dimension K, every codeword is either the zero codeword, or has at most $K-1$ zeros. In this regard, the l-th entries in the k-th coded tiny result block from all $\widetilde{\mathbf{s}}_1, \ldots, \widetilde{\mathbf{s}}_N$ form a codedword of an $[N, K]$ MDS code, and hence contains either all zero vectors, or at most $K-1$ zero vectors (note that a vector is an element in the codeword).

The former case implies that each of the l-th transactions in $b_{k,1}, \ldots, b_{k,K}$ passes verification. The latter case implies that at least one of theses transactions is invalid, and the l-th coded transaction in the k-th coded tiny block of every coded incoming strip must be set to zero before being appended. To simplify the problem, every node i creates a *binary results* vector $g_{*,i}$, which is the i-th column of matrix $\mathbf{G} \in \{0,1\}^{QK \times N}$. Each entry of $g_{*,i}$ is associated with an entry of $\widetilde{\mathbf{s}}_i$; it equals to 0 if the corresponding entry in $\widetilde{\mathbf{s}}_i$ is a zero vector, and equals to 1 otherwise (line 35, Algorithm 2). This operation is encapsulated in the function BINARY, whose pseudocode implementation is omitted for its simplicity. Note that each row of $\mathbf{G}$ is either all zeros, or contains at most $K-1$ zeros. Clearly, the indicator vector g equals to the reduction of all columns of $\mathbf{G}$ with operator bitwise OR.

Let λ be a $(K+f, N)$ threshold signature scheme, and let τ be a $(f+1, N)$ threshold signature scheme. Using the binary results vector $g_{*,i}$, node i obtains a *partial indicator* as the output of the function PARTIALINDICATOR (line 36, Algorithm 2). This function defines a length-QK vector, denoted by gw_i, such that for every $\ell \in [QK]$,

$$gw_i[\ell] = \begin{cases} \langle \ell, 0, \mathit{header} \rangle_{\lambda,i} & g_{\ell,i} = 0 \\ \langle \ell, 1, \mathit{header} \rangle_{\tau,i} & g_{\ell,i} = 1 \end{cases}.$$

Node i sends gw_i back to the leader in the acknowledgment of the PRE-COMMIT message. The leader collects gw_i's from a quorum of $N-f$ nodes and merges them into a length-QK vector gw using function MERGEINDICATORS (line 3, Algorithm 3); the details are given as follows.

Among the ℓ-th entries of the collected vectors $\{gw_j\}_{j \in \mathcal{J}}$, if there exist $K+f$ partial signatures endorsing 0 (generated by the λ scheme), the leader generates and stores a valid signature $\langle \ell, 0, \mathit{header} \rangle_{\lambda}$ in the ℓ-th entry of gw. Otherwise, if there exists $f+1$ partial signatures endorsing 1 (generated by the τ scheme), the leader stores a valid signature $\langle \ell, 1, \mathit{header} \rangle_{\tau}$. Notice that exactly one of these cases must hold due to the following lemma. Note that we implicitly assume that the leader is guaranteed to obtain responses from a quorum $\mathcal{J}$ in the PRE-COMMIT phase; such an assumption will be justified in Theorem 2 on the liveness property of our scheme.

Lemma 2. *Among the ℓ-th entries of the collected vectors $\{gw_j\}_{j \in \mathcal{J}}$ from a quorum of size $|\mathcal{J}| = N-f$, the leader is guaranteed to obtain $K+f$ partial signatures endorsing* 0*, or $f+1$ partial signatures endorsing* 1*, but not both.*

Proof. For any $\ell \in [QK]$, if the ℓ-th row of $\mathbf{G}$ is all-zero, then at least

$$N-2f = (K-1)d + f + 1 \geq (K-1)d + f + 1 - (K-1) + (K-1) = (K-1)(d-1) + f + K \geq f + K$$

vector gw_i's are from correct nodes; they all have zero ℓ-th entry and sign using the λ scheme. Meanwhile, there exist at most f 1's, all from the Byzantine nodes.

If the ℓ-th row of $\mathbf{G}$ is not all-zero, then the number of nodes (at least $N-(K-1)$) having 1's must intersect with the quorum on at least $(N-f) + N - (K-1) - N$ nodes, which equals to

$$N - f - (K-1) = (K-1)d + 2f + 1 - (K-1) = (K-1)(d-1) + 2f + 1 \geq 2f + 1$$

nodes, out of which at least $f+1$ are correct; they all endorse 1 and sign the entry using the τ scheme. Also, there exist at most $(K-1+f)$ 0's, out of which $K-1$ are from correct nodes, and at most f are from Byzantine nodes. ☐

The leader then broadcasts the vector gw to every node in the COMMIT phase. Every node i can learn the indicator vector g from gw, i.e., for every $\ell \in [QK]$,

$$g[\ell] = \begin{cases} 0 & gw[\ell] = \langle \ell, 0, \mathit{header} \rangle_{\lambda} \\ 1 & gw[\ell] = \langle \ell, 1, \mathit{header} \rangle_{\tau} \end{cases}.$$

It then uses the indicator variable to "filter out" invalid transactions in the coded incoming strip $\widetilde{\mathbf{v}}_i$ (line 12, Algorithm 3).

V. DISCUSSION

In this section, we discuss the security, liveness, and the communication complexity aspects of our design. In particular, we investigate the tradeoff between bit complexity and security level.

A. Security

The security level of our scheme is reflected by the upper bound of f compared with N, i.e., the maximum fraction of Byzantine nodes that can be tolerated in the system. The following theorem shows that, for correct verification of transactions, f depends on the total number of nodes N, the number of shards K, and the degree d of the verification function.

Theorem 1. *If $N \geq (K-1) \cdot d + 3f + 1$, our design provides coded consensus.*

Proof. First, HotStuff guarantees safety [20, Theorem 2] (see Section IV for definitions) of the header chain when $N \geq 3f+1$, which is a weaker assumption than $N \geq (K-1)d + 3f + 1$. Note that the added mechanisms are irrelevant to the safety property, as no extra conditions on which nodes can accept a header are introduced. The property of homomorphic fingerprinting function assures the consistency between the coded fragments received by each node [32, Theorem 3.4]. Together, consistency is maintained.

Second, as seen in Lemma 1, our method maintains homology between the coded incoming strips and the coded outgoing strips when $N \geq (K-1) \cdot d + 3f + 1$.

Finally, in order to obtain the indicator vector g, every node needs to decode an $[N, L]$ Reed-Solomon code from $N - f$ elements in the codeword, where $L = (K-1)d + 1$ (see Section IV-D). Since $N - f \geq (K-1)d + 2f + 1$, the property of Reed-Solomon code guarantees correct decoding in this case. Thus, validity is maintained. □

B. Liveness

Although the proposed algorithm provides coded consensus, adversaries may conduct a liveness attack, i.e., prevent the system from processing new transactions. In this section, we show that the proposed algorithm also provides liveness.

Theorem 2. *In the partial synchrony model, the proposed algorithm provides liveness after Global Stabilization Time (GST, see Section II-B).*

Proof. As shown in [20, Theorem 4], HotStuff provides liveness after GST. That is, a decision is reached given that there is a bounded duration T_f, in which all correct nodes remain in the same view, and the view-leader is correct. We show that this property is preserved with the added mechanisms. Specifically, in our modified algorithm, there are precisely three occasions, one in each phase, in which liveness can be affected: line 13, Algorithm 2, line 33, Algorithm 2, and line 10, Algorithm 3. In these occasions, a correct leader might fail to collect sufficiently many responses, and thus liveness might not be guaranteed. We show that each of these occasions depends on a Boolean predicate which is guaranteed to be satisfied when the leader is correct, and thus liveness is preserved.

The first predicate (line 13, Algorithm 2) checks if the received header agrees with the received strips. It is true in every correct node since a correct leader follows the protocol. Therefore, a correct leader is guaranteed to receive valid responses in the PREPARE phase from $N - f$ nodes.

For the second predicate (line 33, Algorithm 2), given the $N - f$ valid responses from the PREPARE phase, a correct leader is able to construct two $(N - f) \times N$ matrices. The j-th row of these matrices will make the green-colored function calls in line 33, Algorithm 2 to return true for node j. For the same reason, a correct leader is able to construct an $(N-f) \times N$ matrix, whose j-th row will make the blue-colored function call in the same line true. Therefore, every correct node will respond in the PRE-COMMIT phase, and hence the correct leader will receive responses, each containing a valid partial signature on *QI*, from $N - f$ nodes.

Finally, the leader is able to generate a valid signature $\langle QI(\mathcal{I}) \rangle_\pi$ on the quorum identifier from the partial signatures. Therefore, the third predicate (line 10, Algorithm 3) is true as well. □

C. Communication Complexity

We analyze the communication complexity for the system to process a block $\mathbf{B}$ that contains $P = QK^2$ transactions, and then compare it to ordinary blockchain designs. The bit complexity of the different stages of our protocol is analyzed next, and sumarized in Table I. Note also that the message complexity is linear thanks to the HotStuff protocol in use.

	PREPARE	PRE-COMMIT	COMMIT	DECIDE
Leader	$O(N \log N + dQK \log N)$	$O(NQ \log N)$	$O(QK)$	$O(1)$
Node	$O(NQ \log N)$	$O(QK)$	$O(1)$	N/A

TABLE I
BIT COMPLEXITIES OF A SINGLE MESSAGE FROM THE LEADER TO A NODE, AND FROM A NODE TO THE LEADER, IN EACH OF THE STAGES.

In the PREPARE phase, the leader sends a checksum and two coded fragments to each of the N nodes. A checksum contains N signatures over $\mathbb{F}_q$, and a coded fragment contains $\frac{|\mathbf{B}|}{K}$ bits. Recall that a block $\mathbf{B}$ contains QK^2 transactions, and each contains a lookup table whose size scales logarithmically with the number of transactions in a shard, same as the degree of the polynomial verification function d (see Section III-B). Further, since the underlying field $\mathbb{F}_q$ must contain at least N distinct elements, it follows that the size of a field element is $O(\log N)$ bits. Together, the size of a block is $O(dQK^2 \log N)$, and the size of a coded strip is $O(dQK \log N)$. Note that the leader also broadcast the header, which contains $2N$ hash values and $2K$ fingerprints, each has a constant number of field elements. Therefore, the message from the leader to a single node in this step is $O(N \log N + dQK \log N)$.

Also in the PREPARE phase, every node i sends N signatures (line 15, Algorithm 2), as well as N coded tiny result blocks (line 17), to the leader. Recall that every coded tiny result block contains Q verification results, each is a length-$(C+E)$ vector, where $C+E$ is the outputs of hash functions and hence constant. Therefore, each message from a node to the leader in the PREPARE phase has a size of $O(N+NQ(C+E)\log N) = O(NQ\log N)$.

In the PRE-COMMIT phase, node i receives $(N-2f)$ signatures (line 25, Algorithm 2) and $N-2f$ coded tiny result blocks (line 26) back from the leader. Therefore, the size of a message from the leader to a node is also $O(N+NQ\log N) = O(NQ\log N)$. Next, still in the PRE-COMMIT phase, every node i sends a partial indicator vector (line 36) to the leader, whose size is $O(QK)$ as it contains QK partial signatures. In the COMMIT phase, every node receives a length-QK vector of threshold signatures (line 3). In addition, every message sent to the leader contains a partial signature and hence has a size of $O(1)$. Similarly, every message sent from the leader in the DECIDE phase contains a threshold signature (in *commitQC*), and hence has size $O(1)$. Together, the bit complexity of our design is as follows.

Corollary 1. *For $\mu < 1/3$, to tolerate μN Byzantine nodes in a system with N nodes, the overall bit complexity for verifying a block of $P = K^2 Q$ transactions is $O(\frac{Pd^2\log N}{(1-3\mu)^2})$.*

Proof. From Table I, the overall bit complexity is

$$O(N^2\log N + dNQK\log N + N^2Q\log N + NQK + N) = O(N^2Q\log N + dNQK\log N).$$

Taking the maximum possible f given the parameter restriction in Theorem 1, we have that $N = (K-1)\cdot d + 3f + 1$, and hence for $n = \frac{(K-1)d}{f}$ we have

$$\frac{N}{K} \approx \frac{N-1}{K-1} = \frac{(3+n)f}{nf/d} = d\left(1+\frac{3}{n}\right), \tag{15}$$

Further, since $\frac{N^2Q\log N}{dNQK\log N} = \frac{N}{Kd} \approx 1 + \frac{3}{n} \geq 1$, it follows that the overall bit complexity is $O(N^2Q\log N)$. As we have $N = (K-1)d + 3f + 1$ by Theorem 1, the system tolerates a fraction $\mu = \frac{f}{N} = \frac{1}{3+n+1/f} \approx \frac{1}{3+n}$ of Byzantine nodes. We can now express the overall bit complexity as a function of μ:

$$O(N^2Q\log N) = O((N/K)^2K^2Q\log N) = O(Pd^2(1+3/n)^2\log N) = O\left(P\frac{d^2\log N}{(1-3\mu)^2}\right). \qquad \square$$

That is, for a system of N nodes and the verification function of degree d, the system designer can choose a value for μ, and the bit complexity for verifying a block scales quadratically with d and logarithmically with N. Note that the degree d scales logarithmically with the number of transactions on one shard. We hereby rewrite the bit complexity for verifying one block as

$$O(P\log^2 M(t)\log N),$$

where $M(t)$ equals to the number of transactions on one shard at epoch t.

To show the novelty of our design, we define the *communication gain* $\mathcal{G}$ as the ratio between the bit complexity common in ordinary blockchain systems, which require every node to receive every transaction, and the bit complexity of our design; the former leads to an inevitable $O(NP)$ bit complexity assuming that each transaction requires a constant amount of bits, and a block contains P transactions. Specifically, if the system in our design tolerates μN Byzantine nodes, where $\mu < \frac{1}{3}$, the communication gain is

$$\mathcal{G} = \frac{NP}{P\frac{d^2\log N}{(1-3\mu)^2}} = \frac{N(1-3\mu)^2}{d^2\log N}. \tag{16}$$

It is evident from (16) that the communication gain is significant for any fixed value of μ and d. Moreover, increasing the number of nodes in the system while keeping the remaining parameters fixed *improves* the overall communication gain with respect to traditional designs; this is a highly desirable property of blockchain systems.

D. Communication-Security Tradeoff

By Corollary 1, the overall bit complexity is $O(P\frac{d^2\log N}{(1-3\mu)^2})$, from which a tradeoff between security and communication is evident. A lower μ value yields low bit complexity, but degrades the security level (since $\mu = f/N$). In contrast, a higher μ value allows the system to tolerate more Byzantine nodes, but inevitably leads to a higher bit complexity. In Figure 3 we illustrate the function $\mu \mapsto \frac{1}{(1-3\mu)^2}$, which describes the tradeoff between μ and the bit communication complexity, measured relative to the baseline $Pd^2\log N$ in Corollary 1.

VI. Future Work and Concluding Remarks

This paper focuses on verifying the validity of new transactions, but does not discuss how nodes can learn if an old transaction has already been redeemed. Directions for future work include incorporating light nodes, and developing algorithms for them

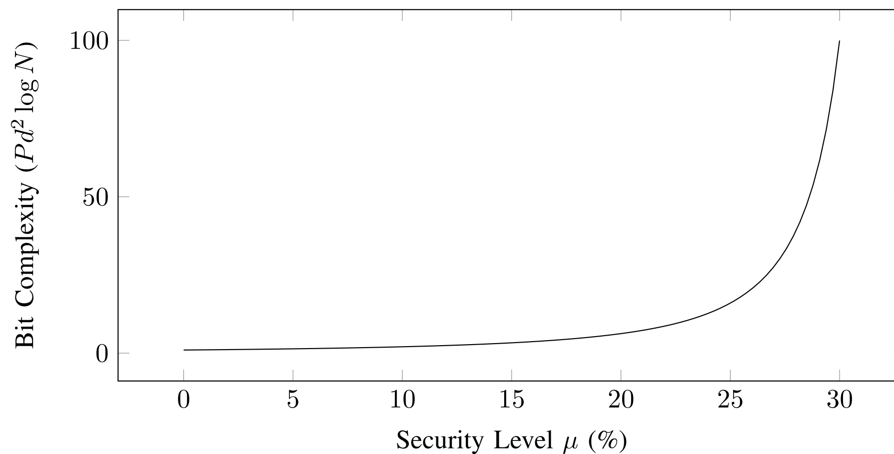

Fig. 3. An illustration of the tradeoff between the security level $\mu = \frac{f}{N}$ and communication bit complexity.

to access raw data by querying a coded distributed system with Byzantine nodes. Finally, as this paper adopts a simplified UTXO model, the generalized multi-input multi-output setting is an interesting direction for future research. In spite of these disadvantages, our work shows that coded computation can alleviate the communication burden in blockchain systems, while maintaining the computations and storage benefits of sharding.

References

[1] C. Wang and N. Raviv, "Low Latency Cross-Shard Transactions in Coded Blockchain," in *IEEE International Symposium on Information Theory*, pp. 2678–2683, 2021.
[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.
[3] K. Croman, et al., "On scaling decentralized blockchains," in *International Conference on Financial Cryptography and Data Security*, pp. 106–125, 2016.
[4] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse, "Bitcoin-NG: A scalable blockchain protocol," *USENIX Symp. Networked Systems Design and Implementation*, pp. 45–59, 2016.
[5] R. Pass and E. Shi, "Hybrid Consensus: Efficient consensus in the permissionless model," *International Symposium on Distributed Computing*, 2017
[6] E. Kokoris-Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing Bitcoin security and performance with strong consistency via collective signing", *USENIX Security Symposium*, pp. 279–296, 2016.
[7] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," *Annu. Int. Cryptology Conference*. pp. 357–388, 2017.
[8] P. Daian, R. Pass, and E. Shi, "Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake," in *International Conference on Financial Cryptography and Data Security*, 2019, pp. 23–41.
[9] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling Byzantine Agreements for Cryptocurrencies", *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 51–68, 2017.
[10] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, vol, 4, no. 3 pp. 382–401, 1982.
[11] N. A. Khooshemehr and M. A. Maddah-Ali, "The Discrepancy Attack on Polyshard-ed Blockchains," in *IEEE Int. Symp. Information Theory*, pp. 2672–2677, 2021.
[12] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications*, vol. 21, no. 7, pp. 558–565, 1978.
[13] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys,* vol. 22, no. 4, pp. 299–319, 1990.
[14] D. Malkhi and M. Reiter, "Byzantine quorum systems," *Distributed Computing*, vol. 11, no. 4, pp. 203–213, 1998.
[15] C. Miguel and L. Barbara, "Practical byzantine fault tolerance," *Symposium on Operating Systems Design and Implementation*, vol. 99, pp. 173–186, 1999.
[16] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," in *ACM SIGOPS Symp. Operating Systems Principles*, pp. 45–58, 2007.
[17] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 BFT protocols," in *Proceedings of the 5th European conference on Computer systems*, pp. 363–376, 2010.
[18] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on BFT consensus," 2017. [Online]. Available: arXiv:1807.04938.

[19] G. G. Gueta, , I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. K. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu, "Sbft: a scalable and decentralized trust infrastructure," in *49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN),* pp. 568–580, 2019.
[20] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "HotStuff: BFT consensus with linearity and responsiveness," *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 347–356. 2019.
[21] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of BFT protocols," in *ACM SIGSAC Conference on Computer and Communications Security*, pp. 31–42. 2016.
[22] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo: Faster asynchronous bft protocols," in *ACM SIGSAC Conf. Computer and Communications Security*, pp. 803–818. 2020.
[23] G. Wang, Z. J. Shi, M. Nixon, and S. Han, "Sok: Sharding on blockchain," *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pp. 41–61, 2019.
[24] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," *ACM SIGSAC Conference on Computer and Communications Security*, pp. 17–30, 2016.
[25] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding," *IEEE Symposium on Security and Privacy* (SP), pp. 583–598, 2018.
[26] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," *ACM SIGSAC Conference on Computer and Communications Security*, pp. 931–948, 2018.
[27] S. Duan, M. K. Reiter, and H. Zhang, "BEAT: Asynchronous BFT made practical," in *ACM SIGSAC Conference on Computer and Communications Security,* pp. 2028–2041, 2018.
[28] C. Cachin, K. Kursawe, F. Petzold and V. Shoup, "Secure and efficient asynchronous broadcast protocols," in *Annu. Int. Cryptology Conference*, pp. 524–541, 2001.
[29] S. Li, M. Yu, C.-S. Yang, A. S. Avestimehr, S. Kannan, and P. Viswanath, "Polyshard: Coded sharding achieves linearly scaling efficiency and security simultaneously," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 249–261, 2020.
[30] Q. Yu, S. Li, N. Raviv, S. M. M. Kalan, M. Soltanolkotabi, and S. A. Avestimehr, "Lagrange coded computing: Optimal design for resiliency, security, and privacy," in *International Conference on Artificial Intelligence and Statistics*, pp. 1215–1225, 2019.
[31] C. Cachin and S. Tessaro, "Asynchronous verifiable information dispersal," in *24th IEEE Symposium on Reliable Distributed Systems (SRDS),* pp. 191–201, 2005.
[32] J. Hendricks, G.R. Ganger, and M. K. Reiter, "Verifying distributed erasure-coded data," *Proceedings of ACM symposium on Principles of distributed computing*, pp. 139–146, 2007.
[33] S. Micali, M. Rabin, and S. Vadhan, "Verifiable random functions," in *40th Annual Symposium on Foundations of Computer Science*, pp. 120–130, 1999.
[34] A. Boldyreva, "Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme,", in *International Workshop on Public Key Cryptography,* pp. 31–46, 2003.
[35] J. Ding and A. Petzoldt, "Current state of multivariate cryptography," *IEEE Security & Privacy*, vol. 15, no. 4, pp. 28–36, 2017.
[36] A. Kipnis, J. Patarin, and L. Goubin, "Unbalanced oil and vinegar signature schemes," in *Int. Conf. Theory and Applications of Cryptographic Techniques*, pp. 206–222, 1999.
[37] A. Petzoldt, M.-S. Chen, B.-Y. Yang, C. Tao, and J. Ding, "Design Principles for HFEv- Based Multivariate Signature Schemes," *International Conference on the Theory and Application of Cryptology and Information Security*, pp. 311–334, 2015.
[38] J. Ding and D. Schmidt, "Rainbow, a new multivariable polynomial signature scheme," *Int. Conf. Applied Cryptography and Network Security*, pp. 164–175, 2005.
[39] A. Gervais, G.O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Čapkun, "On the security and performance of proof of work blockchains," *ACM SIGSAC Conference on Computer and Communications Security*, pp. 3–16, 2016.
[40] B. Applebaum, N. Haramaty-Krasne, Y.I shai, E. Kushilevitz, and V. Vaikuntanathan, "Low-complexity cryptographic hash functions," *Innovations in Theoretical Computer Science Conference* (ITCS), 2017.
[41] J-P. Aumasson and W. Meier, "Analysis of multivariate hash functions" in *International Conference on Information Security and Cryptology*, pp. 309–323, 2007.
[42] J. Ding and B.-Y. Yang, "Multivariates polynomials for hashing," in *International Conference on Information Security and Cryptology*, pp. 358–371, 2007.
[43] M. O. Rabin, "Efficient dispersal of information for security, load balancing, and fault tolerance," *Journal of the ACM (JACM)* 36, no. 2, pp. 335–348, 1989.
[44] M. O. Rabin, "Fingerprinting by random polynomials," *Technical report*, 1981.
[45] M. Bellare and P. Rogaway, "Random oracles are practical: A paradigm for designing efficient protocols," in *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pp. 62–73,1993.
[46] V. Bagaria, S. Kannan, D. Tse, G. Fanti, and P. Viswanath, "Prism: Deconstructing the blockchain to approach physical

limits", in *ACM SIGSAC Conference on Computer and Communications Security*, pp. 585–602, 2019.
[47] M. Al-Bassam, A. Sonnino, V. Buterin, and I. Khoffi, "Fraud and data availability proofs: Detecting invalid blocks in light clients," in *International Conference on Financial Cryptography and Data Security*, pp. 279–298, 2021.
[48] M. Yu, S. Sahraei, S. Li, S. Avestimehr, S. Kannan, and S. Viswanath, "Coded merkle tree: Solving data availability attacks in blockchains," in *International Conference on Financial Cryptography and Data Security*, pp. 114–134, 2020.
[49] C. Dwork, N. A. Lynch, and L. J. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, 1988.
[50] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.
[51] S. Dutta, V. Cadambe, and P. Grover, "Short-dot: Computing large linear transforms distributedly using coded short dot products," *Advances In Neural Information Processing Systems*, 2016.
[52] Q. Yu, M. Maddah-Ali, and S. Avestimehr, "Polynomial codes: an optimal design for high-dimensional coded matrix multiplication," *Advances in Neural Information Processing Systems*, 2017.
[53] R. Tandon, Q. Lei, A. G. Dimakis, and N. Karampatziakis, "Gradient coding: Avoiding stragglers in distributed learning," in Proc. 34th Int. Conf. Mach. Learn. (ICML), Aug. 2017, pp. 3368-–3376.
[54] N. Raviv, I. Tamo, R. Tandon, A. G. Dimakis, "Gradient coding from cyclic MDS codes and expander graphs," IEEE Transactions on Information Theory, vol. 66, no. 12, pp. 7475–7489, 2020.
[55] A. B. Das and A. Ramamoorthy, "Coded sparse matrix computation schemes that leverage partial stragglers," *IEEE Transactions on Information Theory*, 2022.
[56] C.-S. Yang, R. Pedarsani, and A. S. Avestimehr, "Timely coded computing," in *IEEE International Symposium on Information Theory (ISIT)*, pp. 2798–2802, 2019.
[57] N. Woolsey, R.-R. Chen, and M. Ji, "Cascaded coded distributed computing on heterogeneous networks," in *IEEE International Symposium on Information Theory (ISIT)*, pp. 2644–2648, 2019.
[58] S. Li, S. Sahraei, M. Yu, S. Avestimehr, S. Kannan, and P. Viswanath, "Coded State Machine–Scaling State Machine Execution under Byzantine Faults," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pp. 150–152, 2019.