Hyra: Scalable Byzantine-Resilient State Storage Engine with Hierarchical Erasure-Coding

QIFENG QUE, East China Normal University, China and Engineering Research Center of Blockchain Data Management, Ministry of Education, China
XIAODONG QI, Nanyang Technological University, Singapore
ZHAO ZHANG*, East China Normal University, China and Engineering Research Center of Blockchain Data Management, Ministry of Education, China
YANQIN YANG*, East China Normal University, China and Engineering Research Center of Blockchain Data Management, Ministry of Education, China
CHEQING JIN, East China Normal University, China and Engineering Research Center of Blockchain Data Management, Ministry of Education, China
AOYING ZHOU, East China Normal University, China and Engineering Research Center of Blockchain Data Management, Ministry of Education, China

Account-based blockchains maintain evolving state data, such as account balances, across all replicas to ensure consistency. However, the traditional full-replication model incurs significant storage overhead and limits scalability, especially as transaction throughput increases. While partitioning is a natural solution, applying it to blockchain state is challenging due to structural dependencies, verifiability requirements, and the presence of Byzantine faults. We present Hyra, a scalable state storage engine that enables fault-tolerant and efficient state partitioning. Hyra introduces: (i) locality-aware partitioning with embedded indexing to preserve access efficiency; (ii) hierarchical erasure coding with sub-chunk recovery to minimize decoding overhead; and (iii) a hybrid verification mechanism that combines vector commitments and Merkle trees for fine-grained integrity checking. Our design achieves provably optimal redundancy while maintaining constant per-replica storage overhead. Experiments show that Hyra reduces storage by up to 92.3% without compromising performance.

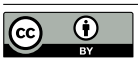
CCS Concepts: • **Information systems** → *Data management systems.*

Additional Key Words and Phrases: Blockchain, state storage, erasure coding, fault tolerance

ACM Reference Format:
Qifeng Que, Xiaodong Qi, Zhao Zhang, Yanqin Yang, Cheqing Jin, and Aoying Zhou. 2026. Hyra: Scalable Byzantine-Resilient State Storage Engine with Hierarchical Erasure-Coding. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 44 (February 2026), 27 pages. https://doi.org/10.1145/3786658

*Corresponding author.

Authors' Contact Information: Qifeng Que, East China Normal University, Shanghai, China and Engineering Research Center of Blockchain Data Management, Ministry of Education, Shanghai, China, qfque@stu.ecnu.edu.cn; Xiaodong Qi, Nanyang Technological University, Singapore, Singapore, xiaodong.qi@ntu.edu.sg; Zhao Zhang, East China Normal University, Shanghai, China and Engineering Research Center of Blockchain Data Management, Ministry of Education, Shanghai, China, zhzhang@dase.ecnu.edu.cn; Yanqin Yang, East China Normal University, Shanghai, China and Engineering Research Center of Blockchain Data Management, Ministry of Education, Shanghai, China, yqyang@dase.ecnu.edu.cn; Cheqing Jin, East China Normal University, Shanghai, China and Engineering Research Center of Blockchain Data Management, Ministry of Education, Shanghai, China, cqjin@dase.ecnu.edu.cn; Aoying Zhou, East China Normal University, Shanghai, China and Engineering Research Center of Blockchain Data Management, Ministry of Education, Shanghai, China, ayzhou@dase.ecnu.edu.cn.



ACM 2836-6573/2026/2-ART44
https://doi.org/10.1145/3786658

1 Introduction

Smart contracts execute predefined logic by reading and modifying *state data*, which tracks the dynamic status of on-chain variables (e.g., account balances). As each block is committed, the states are updated according to included transactions. To efficiently manage and verify these updates, blockchains organize the states into a tree-based authenticated data structure, the *state tree* [30, 40, 43, 65], which integrates indexing and Merkle proofs for integrity verification, as shown in Fig. 1. To preserve decentralization, most blockchains adopt a *full-replication* model, where every node (or *replica*) stores the complete global state. However, this model limits scalability since the per-replica storage cost remains constant as the network grows. For instance, Solana [7] processes about 3,000 transactions per second, increasing state data by roughly 2.8 GB per day[1]. With advances in consensus [42, 47, 49, 70] and parallel execution [19, 29, 33], this storage burden continues to escalate.

To improve scalability, a natural solution is *data partitioning*, a widely adopted technique in distributed systems [21, 26, 63] that spreads data across multiple replicas. However, this approach is vulnerable under Byzantine faults, where malicious replicas may drop or withhold data. To address this, prior works [23, 53, 55, 56, 67] combine Byzantine fault tolerance (BFT) with *erasure coding* (EC), which splits data into redundant fragments that enable recovery from any sufficiently large subset. By tuning EC parameters, these systems tolerate replica faults while ensuring data availability under failures.

However, these approaches are designed for block data, which is flat and append-only, and thus do not extend naturally to structured state data organized as a tree. To apply erasure coding, tree nodes must be serialized into *chunks*, the basic units for encoding and distribution. This process must preserve parent-child relationships and support efficient, deterministic node-to-chunk and chunk-to-replica mapping to enable indexable access without centralized coordination. Retrieving a state value requires traversing the state tree from root to leaf based on the requested key. Once the relevant nodes distributed across replicas, the process often involves accessing multiple replicas, resulting in high communication overhead.

If any node becomes unavailable due to faulty or malicious replicas, the system employs erasure decoding to reconstruct it. However, because the encoded units are coarse-grained chunks, the system must decode the entire chunk even when only a small portion, such as a single node, is needed. This leads to decoding amplification and increased latency. Additionally, in existing approaches, the size of the data batch to be encoded grows with the number of replicas, further prolonging the decoding process. Moreover, while EC ensures recoverability from replica failures, it does not guarantee the integrity of the retrieved data. Faulty replicas may return tampered chunks. Attaching a hash to each chunk is a simple mitigation, but it introduces extra storage overhead and only enables coarse-grained verification. This approach is insufficient to ensure the correctness of individual nodes within a chunk.

Our Solution. To address the above issues, we propose *Hyra*, an EC-based state storage engine that enables scalable and resilient state partitioning. Hyra is designed to ensure high recoverability under Byzantine failures while maintaining efficient read performance. Its main contributions are summarized as follows:

- **Locality-aware partitioning and embedded indexing.** Hyra partitions the state tree in a bottom-up fashion and embeds lightweight metadata in parent nodes to reference the locations of their children, preserving logical relationships across distributed chunks without relying on centralized indexing.

[1]https://explorer.solana.com/

- **Hierarchical encoding and sub-chunk recovery.** To reduce and balance recovery overhead, Hyra adopts a hierarchical EC strategy that enables incremental decoding from low-latency groups to high-tolerance ones. Chunks are further divided into fixed-size sub-chunks for fine-grained recovery and reduced communication cost.
- **Hybrid data verification.** Hyra adopts a hybrid verification mechanism combining *Vector Commitments* (VC) and *Merkle trees*. Each chunk is structured as a Merkle tree over its sub-chunks, with root hashes committed via a VC. This enables node-level integrity checks with minimal proof size and storage overhead.
- **Formally optimal coding scheme.** Hyra employs a provably optimal coding configuration under BFT constraints, which minimizes redundancy while ensuring recoverability. Compared to conventional BFT-based approaches, it reduces storage overhead by a factor of f/n, achieving better scalability.

Finally, we integrate Hyra into the official Ethereum implementation, then evaluate its storage overhead and read performance. Our results show that Hyra significantly reduces storage overhead while having minimal impact on reading efficiency.

Organization. The remainder of this paper is organized as follows. Section 2 provides the necessary background. Section 3 presents an overview of the Hyra architecture, and Section 4 details the system design. Section 5 offers theoretical analysis, while Section 6 reports the experimental results. We discuss related work in Section 7, and conclude the paper in Section 8.

2 Background and Preliminaries

This section provides the necessary background and key concepts required for the rest of the paper.

2.1 Blockchain

We begin with key components of blockchain systems, including account models, state representation, and consensus mechanisms.

Account Models and State Representation. Blockchain platforms typically employ one of two account models: the *UTXO* (Unspent Transaction Output) model [51] and the *account-based* model [65]. The UTXO model, as used in Bitcoin, structures transactions as sets of inputs and outputs, where each input consumes a prior unspent output, and each output becomes a new spendable unit. In contrast, Ethereum adopts an account-based model, in which each address maintains a mutable *state* (e.g., balance), updated directly during execution. This model simplifies the deployment of *smart contracts*, self-executing programs that enforce rules embedded in transactions. State represents application-level data such as balances or asset ownership, typically structured as key-value mappings that evolve incrementally with each transaction[2]. To efficiently manage and verify dynamic state, most blockchain systems leverage Merkle trees [45] or their variants [5, 6, 34, 65], as detailed in Section 2.3. All state transitions are recorded immutably, ensuring transparency, consistency, and traceability.

Consensus. Consensus ensures that a set of distributed replicas agrees on the next valid block to append to the ledger, even in the presence of up to f Byzantine (malicious) replicas. Hyra adopts a PBFT-like protocol [16], commonly used in permissioned blockchains [2, 3, 60]. In each consensus round, a designated primary replica (the leader) proposes a block and broadcasts it to all other replicas. Upon validating the proposal, each replica sends a `Prepare` message. Once a replica receives at least $n-f$ `Prepare` messages, it enters the commit phase and broadcasts a `Commit` message. A block is finalized when a replica gathers $n-f$ `Commit` messages, forming a *quorum*.

[2]We use "execute smart contracts" and "execute transactions" interchangeably throughout this paper.

The result is then executed and returned to the client. PBFT assumes a static honesty model and guarantees safety and liveness as long as $n \geq 3f+1$.

2.2 Cryptography

Cryptographic hash function. A cryptographic hash function $\mathsf{H} : \{0,1\}^* \rightarrow \{0,1\}^\lambda$ is a deterministic algorithm that maps inputs of arbitrary length to fixed-length outputs (called digests). It provides three essential security properties:

- *Collision resistance*: it is computationally infeasible to find two distinct inputs $x \neq x'$ such that $\mathsf{H}(x) = \mathsf{H}(x')$.
- *Preimage resistance*: given a hash output y, it is infeasible to find any input x such that $\mathsf{H}(x) = y$.
- *Second-preimage resistance*: given an input x, it is hard to find a different input $x' \neq x$ such that $\mathsf{H}(x) = \mathsf{H}(x')$.

These properties ensure that even a slight change in the input yields a completely different and unpredictable output, making hash functions fundamental to data integrity.

Merkle trees. A Merkle tree is a hash-based authenticated data structure enabling efficient commitment to an ordered sequence of data items $V = \langle v_0, \ldots, v_{q-1} \rangle$. It is constructed as follows:

- $\mathsf{Merkle}(V)$ builds a binary hash tree $\mathcal{T}$ where each leaf is $d_i = \mathsf{H}(v_i)$, and each internal node is computed as $\mathsf{H}(d_l || d_r)$, with d_l and d_r being the hashes of the left and right children, respectively. The root digest θ serves as a succinct commitment to the entire sequence.
- $\mathsf{Prove}(\mathcal{T}, i)$ generates a membership proof for v_i as a Merkle path $\pi_i = \langle (d_0, \mathsf{L/R}), \ldots, (d_{\ell-1}, \mathsf{L/R}) \rangle$. Starting from the leaf node $d_i = \mathsf{H}(v_i)$, the path records, at each of the ℓ levels toward the root, the hash d_j of the sibling node and its position $\mathsf{L/R}$ indicating whether it is a left or right sibling.
- $\mathsf{MVerify}(\theta, \pi_i, v_i, i)$ reconstructs a root digest θ' from v_i and its Merkle path π_i by iteratively applying H according to the specified sibling positions. The proof is accepted if $\theta' = \theta$, confirming the validity of v_i under the committed root.

Vector Commitment. A vector commitment (VC) [28, 41] is a cryptographic primitive that enables a compact commitment to an ordered sequence, while supporting efficient, index-specific verification. A VC exposes the following polynomial-time algorithms:

- $\mathsf{Commit}(V)$ takes a sequence $V = \langle v_0, \ldots, v_{q-1} \rangle$ and outputs a commitment Φ.
- $\mathsf{Open}(V, \Phi, i)$ generates a fixed-length proof Π_i attesting that v_i is the committed value at index i.
- $\mathsf{Verify}(\Phi, \Pi_i, v_i, i)$ checks whether v_i is indeed the committed value at position i, returning `true` iff the proof is valid.

A secure VC should guarantee two key properties:

- *Succinctness*: both the commitment Φ and each proof Π_i are of constant size $O(1)$ with respect to the sequence length q.
- *Position binding*: for any index i, it is computationally infeasible to produce two distinct values $v \neq v'$ with valid proofs under the same commitment Φ.

2.3 State Trees

State data reflects the dynamic system status and must support three key properties: (1) *indexability* for efficient access and updates, (2) *verifiability* for cryptographic integrity checks, and (3) *traceability* for historical inspection. To meet these requirements, blockchains typically organize state as an authenticated index tree, known as the *state tree* [30, 40, 43, 65], incorporating Merkle proofs [45]. This work adopts the Merkle Patricia Trie (MPT), as used in Ethereum, as the representative state structure.

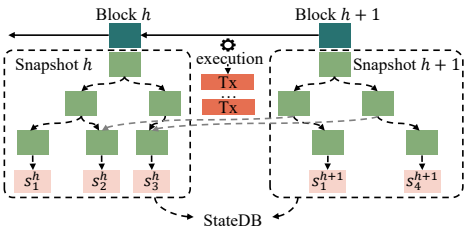


Fig. 1. Example of state storage and update

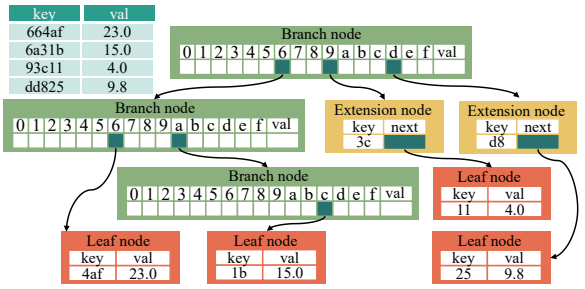


Fig. 2. Example of Merkle Patricia Trie

In an MPT as depicted in Fig. 2, each state entry is indexed by a key, which is divided into sequential 4-bit segments, called *nibbles*, to support path-based navigation. The trie contains three types of nodes: (1) A *branch* node has up to 17 children, where the first 16 represent possible nibble values (0–f), and the 17th stores a value if the key terminates at that node. (2) A *leaf* node holds a value along with the remaining unmatched nibbles, marking the end of a key path. (3) An *extension* node stores a shared nibble prefix, used to compress paths without branching.

Indexability. To retrieve the value of a state with a key κ, the MPT is traversed from the root to a leaf node, guided by the sequence of nibbles in κ. At each branch node, the current nibble determines which child to visit next. Upon reaching a leaf node, the remaining unmatched nibbles in the key are compared against the encoded path stored in the leaf. If they match, the corresponding value v is returned; otherwise, the key κ does not exist in the trie.

Verifiability. To verify a state value v with a key κ, the MPT generates a proof consisting of all nodes along the search path from the root to the corresponding leaf that accommodates v. Each inner node stores the cryptographic hash of its children, enabling the verifier to recompute the root hash by hashing the nodes in the proof. If the recomputed root matches the known root hash, the value v is deemed valid. This root hash is included in each block header as a digest summarizing the global state.

Traceability. The MPT stores each node as a separate record in a key-value database (e.g., LevelDB [48, 52]), using the node's hash as the key and its serialized content as the value. To support traceability, MPT employs a *copy-on-write* mechanism: modified nodes are copied and updated. Therefore, new keys derived from their updated hashes, and all historical versions will be preserved in the key-value database. This allows the exact state at any block height to be reconstructed from the corresponding root.

2.4 Storage Partition

To preserve decentralization, most blockchains adopt full replication, where every replica stores all blocks and state data. While this ensures strong fault tolerance and security, it leads to constant per-replica storage cost, independent of network size. As consensus protocols [25, 37, 38] and execution engines [11, 13, 54] improve throughput, more frequent state updates further intensify storage pressure. To address this, recent proposals suggest partitioning blockchain data across replicas, which is effective in conventional distributed systems [12, 36], but faces two key challenges in blockchain. First, Byzantine replicas may tamper with partitions they hold, jeopardizing data correctness. Second, ensuring availability still requires replicating each partition across many nodes (e.g., 33% in PBFT), diminishing the expected storage savings.

BFT-Store [55] addresses Byzantine-resilient storage by combining *erasure coding* (EC) with BFT consensus. A (K, M) EC scheme encodes K data chunks into M parity chunks, tolerating up to M failures by enabling recovery from any K out of $K + M$ chunks. Reed–Solomon coding, widely used

for EC, performs encoding via matrix multiplication:

$$\begin{bmatrix} 1 & \alpha_1^1 & \dots & \alpha_1^{K-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_M^1 & \dots & \alpha_M^{K-1} \end{bmatrix} \times \begin{bmatrix} D_1 \\ \vdots \\ D_K \end{bmatrix} = \begin{bmatrix} P_1 \\ \vdots \\ P_M \end{bmatrix} \tag{1}$$

Here, D_i and P_j are data and parity chunks in a *coding group*.

Specifically, BFT-Store adopts an $(n-2f, 2f)$ erasure coding scheme, enabling data recovery as long as chunks from at least $n-2f$ honest replicas are available. This reduces the per-block storage complexity from $O(n)$ to $O(1)$, significantly improving scalability. However, BFT-Store is designed for ledger data (i.e., blocks), which are flat and easily verifiable via a single hash. In contrast, state data is organized as a hierarchical tree with path-dependent verification. Applying BFT-Store directly to state management would flatten the state tree into a monolithic object, eliminating the structural indexing essential for efficient queries and fine-grained verifiable access. Moreover, preserving parent-child relationships within flat, encoded chunks is nontrivial and demands careful handling to maintain logical consistency and retrieval efficiency. One naïve approach is to record node-to-chunk mappings at each replica, but this introduces additional metadata overhead.

3 Overview

In this section, we present an overview of Hyra.

System model. Hyra targets a permissioned blockchain system consisting of n replicas $R_0 \sim R_{n-1}$, where membership is relatively stable and joining replicas must undergo identity verification. It follows the standard Byzantine fault-tolerant (BFT) model [16], tolerating up to f Byzantine replicas under the assumption $n \geq 3f + 1$. Honest replicas follow the protocol faithfully and never become faulty. A trusted public key infrastructure (PKI) authenticates identities, and all communication is conducted over authenticated channels. The system operates under a partially synchronous network model [24], where message delays may initially be unbounded but become bounded after an unknown stabilization period, reflecting real-world conditions with network faults. Hyra adopts the account-based model and maintains a state tree to organize global state. While agnostic to the specific tree implementation, Hyra assumes it satisfies three properties: *indexability*, *verifiability*, and *traceability*, as defined in Section 2.3. In this work, we instantiate Hyra with the Merkle Patricia Trie (MPT), as used in Ethereum.

Workflow. Hyra follows the standard blockchain transaction lifecycle: clients submit transactions, which are ordered and finalized through consensus and then executed to update the global state. On top of this, Hyra introduces two new modules per replica: an *encoder*, which processes updated state data, and a *decoder*, which handles state reading and recovery (see Fig. 3).

- At each block height, the encoder collects all modified state tree nodes and organizes them into data chunks. These chunks are encoded via a *multi-level hierarchical erasure coding* scheme: lower levels support lightweight, incremental recovery, while the top level ensures fault tolerance against Byzantine replicas. Both data and parity chunks are then distributed to replicas in a *decentralized* and *verifiable* manner.
- The decoder enables efficient *state reading*. Upon receiving a client query for a key at a given block height, a coordinator replica traverses the state tree by querying the appropriate *target replicas*. If any node is missing or fails verification, the coordinator initiates a decoding process, escalating through coding levels until the node is recovered. Since Reed-Solomon (RS) coding does not inherently verify data correctness, Hyra includes an additional integrity mechanism to authenticate chunks.

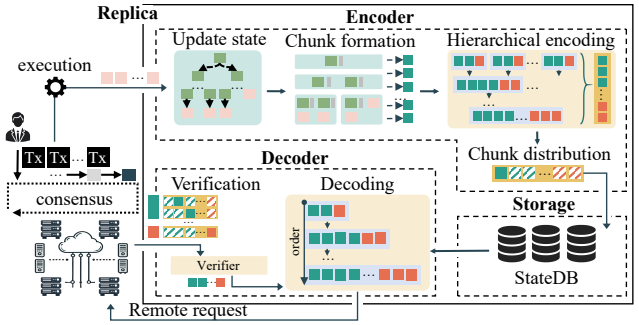


Fig. 3. Overall architecture of Hyra

Partitioning principle. Executing transactions over a fully distributed state would introduce significant overhead. To avoid this, each replica maintains a complete local copy of the latest global state, representing a consistent snapshot of all accounts and contracts at the most recent block height. Historical state nodes, produced via copy-on-write at each block height, are partitioned and distributed across replicas. All state data, including both recent and historical nodes, is persisted in an underlying key-value store.

Challenges. While enabling scalable state storage, Hyra must overcome several challenges. First, due to structural dependencies among nodes, Hyra must preserve parent-child relationships during partitioning and enable efficient node-to-replica resolution during reading. Second, ensuring recoverability under Byzantine faults requires a carefully chosen erasure coding scheme that minimizes redundancy while tolerating faults. An inappropriate configuration may result in data loss if too few honest replicas respond. Third, chunk distribution must be fully decentralized, and an external verification mechanism is needed since erasure coding provides no guarantees of integrity.

4 Design

We now present the design of Hyra, which addresses the key challenges outlined in Section 3.

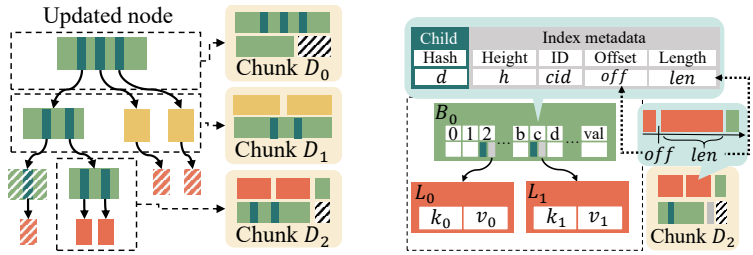
4.1 Chunk Formation

Node partitioning. Hyra partitions the modified state data at each block height h into a collection of *data chunks*. Because nodes in the state tree are independently copied and updated during transaction execution, Hyra identifies all nodes modified at height h and groups them into chunks $\mathcal{D}^h = \langle D_0^h, \ldots, D_{K-1}^h \rangle$[3], as illustrated in Fig. 4a. These chunks are subsequently encoded using an erasure coding scheme to generate a fixed number of *parity chunks*, as elaborated in the next subsection. All resulting chunks, both data and parity, are then distributed across replicas for decentralized storage.

To ensure recoverability under Byzantine faults, Hyra selects a coding configuration compatible with the underlying BFT protocol. Additionally, to improve read performance, Hyra employs a *locality-aware partitioning* strategy: nodes that are frequently accessed together (e.g., along a common path in the state tree) are grouped into the same chunk. This optimization allows multiple related nodes to be retrieved in a single request, significantly reducing cross-replica communication during read operations. We elaborate on this strategy in Section 4.5.

Index metadata embedding. Since nodes are partitioned into chunks and distributed across replicas, Hyra requires an efficient mechanism to locate the chunk containing a specific node during state access. A naive solution is to replicate global index metadata across all replicas, but this approach compromises storage scalability. Instead, Hyra embeds the necessary index metadata

[3]For brevity, the superscript h is omitted when the context is clear.



(a) Node partitioning (b) Index metadata embedding

Fig. 4. Illustration of chunk formation.

directly into the parent nodes, which are themselves distributed, as illustrated in Fig. 4b. For each child node, its parent maintains the following metadata:

- **Height** h: the block height at which the child node was last copied and updated;
- **Chunk ID** cid: the identifier of the chunk containing the node;
- **Offset** off: the byte offset of the node within the chunk;
- **Length** len: the size of the serialized node.

This metadata enables Hyra to locate any node globally: the height identifies the corresponding coding group, the chunk ID selects the specific chunk within that group, and the offset and length determine how to extract the node's content. The metadata for the root node is stored locally by each replica and serves as the entry point for tree traversal. Importantly, Hyra avoids maintaining an explicit chunk-to-replica mapping; instead, it deterministically derives the mapping at read time, as detailed in Section 4.3.

Chunk Layout. In Hyra, nodes in the state tree cannot be serialized into chunks in arbitrary order, as each parent node must include metadata referencing the chunk locations of its children. To address this dependency, Hyra constructs data chunks in a bottom-up manner: child nodes are inserted and assigned to chunks first, and only after all children of a parent node have been placed can the parent's metadata be completed. This process recurses from the leaves to the root of the state tree, finalizing the chunk layout as metadata is populated during node placement. For example, in Fig. 4b, L_1 is a leaf node prioritized for insertion. After L_1 is placed into a chunk, Hyra derives its metadata and embeds it into its parent B_0, which is then serialized and placed. The same procedure applies recursively up the tree until the root is reached. As required by erasure coding, all data chunks within the same coding group must be of equal length. Therefore, after chunk formation, shorter chunks are padded to match the length of the longest chunk before encoding.

Sub-chunks. When a node becomes unavailable, Hyra restores it by decoding the chunk that contains it. However, this approach reconstructs the entire chunk even when only a small portion is required. To improve efficiency, Hyra refines the decoding granularity by dividing each chunk D_i into s fixed-size *sub-chunks*, denoted as $\langle D_{i,0}, \ldots, D_{i,s-1} \rangle$. For a coding group comprising K data chunks, the j-th sub-chunks from all data chunks, $\langle D_{0,j}, \ldots, D_{K-1,j} \rangle$, collectively form the j-th *stripe*. Each stripe serves as the atomic unit of decoding. To recover a node, Hyra determines the stripe range α to β that overlaps with the node, based on its offset and length recorded in the parent node's metadata. It then retrieves only the corresponding sub-chunks within stripes α to β across the coding group. This *stripe-based recovery* mechanism eliminates the need for full-chunk reconstruction, substantially reducing both bandwidth consumption and decoding cost.

4.2 Hierarchical Encoding

While stripe-based decoding improves node recovery performance, it incurs significant communication overhead, especially as the coding group size grows with the number of replicas. To mitigate

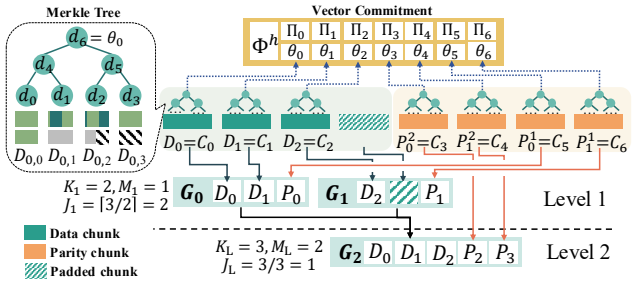


Fig. 5. Example of hierarchical encoding

this, Hyra adopts a *hierarchical encoding strategy* that enables lightweight, incremental recovery with varying levels of fault tolerance. For clarity, we first describe the scheme at the chunk level; extension to sub-chunk recovery is straightforward.

Encoding. Given a set of data chunks $\mathcal{D} = \langle D_0, \ldots, D_{K-1} \rangle$, Hyra applies erasure coding across L levels, as illustrated in Fig. 5. At each level $\ell \in [1, L]$, the encoder partitions $\mathcal{D}$ into groups of K_ℓ data chunks. Each group is encoded to produce M_ℓ parity chunks, resulting in $J_\ell = \lceil K/K_\ell \rceil$ coding groups. If the final group is incomplete, it is padded with empty chunks. The total number of parity chunks at level ℓ is $\lambda_\ell = J_\ell \cdot M_\ell$, denoted by the set $\mathcal{P}^\ell = \{P_0^\ell, \ldots, P_{\lambda_\ell - 1}^\ell\}$. The i-th coding group at level ℓ is encoded as:

$$P_{iM_\ell}^\ell, \ldots, P_{(i+1)M_\ell - 1}^\ell \leftarrow \mathsf{Encode}\left(D_{iK_\ell}, \ldots, D_{(i+1)K_\ell - 1}\right)$$

To form a hierarchical structure, the group size K_ℓ increases with level ℓ. At the top level ($\ell = L$), the entire set $\mathcal{D}$ is treated as a single group ($K_L = K$), providing the strongest protection against data loss. After encoding, all data and parity chunks are distributed across replicas, with each chunk assigned to a unique replica. During recovery, Hyra first attempts decoding at the lowest level ($\ell = 1$) and progressively escalates to higher levels if recovery fails.

As illustrated in Fig. 5, the example adopts two levels of encoding. At the lower level ($\ell = 1$), each group consists of $K_1 = 2$ data chunks and produces $M_1 = 1$ parity chunk, enabling recovery from one missing chunk per group. At the higher level ($\ell = 2$), all data chunks are jointly encoded to provide global redundancy, tolerating up to two missing chunks. Since the last coding group G_1 at level 1 contains only one data chunk, an extra chunk is padded to complete the group. With this setting, any two of the three chunks in a level-1 group suffice for recovery. Assuming that 25% of replicas may be Byzantine, the probability of successful recovery at level 1 reaches 84.3%. In most cases, recovery completes at level 1. If decoding fails due to corrupted or unavailable chunks from malicious replicas, Hyra escalates the recovery to the next level. At the final level, the coding parameters and chunk distribution are configured to ensure that recovery succeeds using responses from at least $n-f$ replicas, guaranteeing robustness against up to f Byzantine faults.

RS Scheme Configuration. In the lower levels (1 to $L-1$), the primary objective is to support lightweight recovery. These levels offer progressively stronger fault tolerance, and their coding parameters (K_ℓ, M_ℓ) can be flexibly chosen, provided that the group size increases with the level index ℓ. In contrast, the top level ($\ell = L$) plays a critical role in ensuring recoverability under Byzantine conditions. Its coding configuration (K_L, M_L) must be carefully designed to guarantee recovery despite adversarial failures.

Existing BFT-coded storage systems [55] typically employ a $(n-2f, 2f)$ scheme, where $n-2f$ data chunks are encoded into $2f$ parity chunks, yielding n total chunks distributed across all n replicas. Since any quorum committing a block must contain at least $n-f$ replicas, and at most f may be Byzantine, this ensures that at least $n-2f$ honest replicas can supply valid chunks for recovery. Hyra adopts a tighter configuration of $(n-2f, f)$ to reduce storage overhead. In this

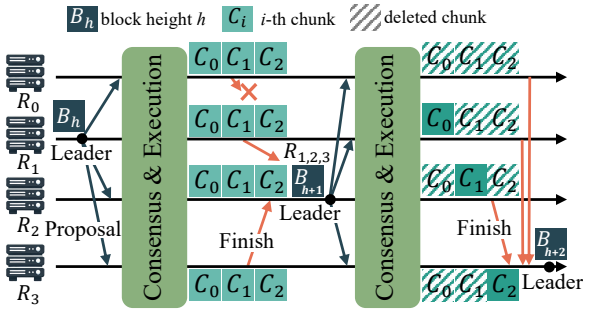


Fig. 6. Workflow of chunk distribution

scheme, $n-2f$ data chunks are encoded into f parity chunks, resulting in a total of $n-f$ chunks. These are distributed across a selected subset of $n-f$ replicas, denoted by $\mathfrak{R}_h$ at block height h. To guarantee successful recovery, it must ensure that at least $n-2f$ of the replicas in $\mathfrak{R}_h$ are honest, thus tolerating up to f Byzantine replicas within this group.

Deterministic Replica Selection. The $(n-2f, f)$ coding scheme used in Hyra requires that all replicas deterministically agree on the same replica set $\mathfrak{R}_h$ at each block height h, without relying on central coordination. Any inconsistency in this selection can lead to over-replication or worse, data loss. For example, in a network of four replicas $R_0 \sim R_3$, suppose R_0, R_1, R_2 compute $\mathfrak{R}_h = \{R_0, R_1, R_2\}$, while R_3 computes $\mathfrak{R}_h = \{R_1, R_2, R_3\}$. As a result, all four replicas may store chunks, violating the $n-f$ constraint and increasing storage overhead. More critically, suppose $R_0 \sim R_2$ agree on $\mathfrak{R}_h = \{R_1, R_2, R_3\}$, while R_3 computes $\mathfrak{R}_h = \{R_0, R_1, R_2\}$. In this case, only R_1 and R_2 actually store chunks (the intersection of the two sets). If R_1 is Byzantine and withholds its data, recovery becomes impossible, despite the system believing enough replicas participated.

To ensure consistency in chunk assignment, Hyra must guarantee agreement on the replica set $\mathfrak{R}_h$ across all replicas. While this could be achieved through a dedicated commit protocol, doing so in a BFT setting would incur communication costs comparable to block consensus itself. To mitigate this overhead, Hyra piggybacks this process onto the existing PBFT commit phase of blocks. PBFT is a three-phase protocol with two rounds of voting, where a replica proceeds to the next phase after receiving at least $n-f$ votes (including its own). A straightforward approach is to select the $n-f$ replicas that participated in the commit phase as $\mathfrak{R}_h = \{R_{t_0}, \ldots, R_{t_{n-f-1}}\}$, ordered by replica ID. However, due to network asynchrony or Byzantine behavior, replicas may observe different subsets of votes, resulting in inconsistent $\mathfrak{R}_h$. To address this, Hyra introduces a lightweight, deterministic selection mechanism, illustrated in Fig. 6. After the commit phase at block height h, each replica sends a signed `Finish` message to the leader. The leader aggregates $n-f$ valid signatures and embeds them into the header of block B_{h+1}. Once B_{h+1} is finalized, these signers collectively define the agreed replica set $\mathfrak{R}_h$ for storing chunks at height h.

To reduce signature overhead, Hyra leverages CoSi [62] to aggregate the $n-f$ signatures into a single compact signature, accompanied by a bitmap identifying the participating replicas. Each replica can independently verify the result by reconstructing the aggregated public key from the bitmap and validating the CoSi signature. The bitmap thus serves as an authenticated, globally agreed representation of $\mathfrak{R}_h$. To avoid consistently including the leader in $\mathfrak{R}_h$, which may lead to storage imbalance, Hyra rotates the leader at each block height in a round-robin manner, following Tendermint's design [15].

Although the `Finish` message introduces an additional round of communication, it incurs negligible bandwidth and latency overhead. This is due to two reasons: (1) the message payload is minimal, and (2) the communication occurs in parallel with transaction execution after the commit phase. We further confirm this conclusion through empirical evaluation in our experiments.

4.3 Chunk Distribution and Verification

With the encoding mechanism in place, the remaining challenge is to reliably distribute chunks across replicas in the presence of adversarial behavior.

Chunk Distribution. Hyra adopts a fully decentralized strategy for chunk distribution. Each replica independently performs encoding across all levels and deterministically retains only the chunks it is responsible for, discarding the rest. The mapping from chunks to replicas is computed locally using a shared, deterministic rule, ensuring consistent assignment across the network without requiring coordination. As the top-level coding group (L) is critical for recoverability, we first describe how its chunks are distributed.

Given the data chunks $\mathcal{D}^h = \langle D_0, \ldots, D_{n-2f-1} \rangle$ and the parity chunks across all levels $\{\mathcal{P}^\ell = \langle P^\ell_0, \ldots, P^\ell_{\lambda_\ell - 1} \rangle \mid \ell = 1, \ldots, L\}$, Hyra merges them into a unified chunk sequence, as shown in Fig. 5:

$$C = \langle \mathcal{D}^h, \mathcal{P}^L, \ldots, \mathcal{P}^1 \rangle = \langle C_0, \ldots, C_{q-1} \rangle, \quad q = n{-}2f + \sum_{\ell=1}^{L} \lambda_\ell$$

In particular, the first $n{-}f$ chunks are derived from the level-L coding group:

$$C_i = \begin{cases} D_i & \text{for } 0 \le i < n{-}2f \\ P^L_{i-(n-2f)} & \text{for } n{-}2f \le i < n{-}f \end{cases}$$

To assign these $n{-}f$ chunks to the $n{-}f$ replicas in $\mathfrak{N}_h$, each replica uses the block hash of B_h to generate a shared permutation σ over $\{0, \ldots, n{-}f{-}1\}$. The i-th chunk C_i is then assigned to the $\sigma(i)$-th replica in $\mathfrak{N}_h$, also referred to as its *target replica*. Since all replicas compute σ deterministically, they independently identify their assigned chunks without coordination, ensuring collision-free and consistent distribution with at least $n{-}2f$ honest holders.

For lower-level parity chunks ($\ell < L$), which optimize read performance rather than recovery guarantees, Hyra applies a lightweight randomized assignment. Each such chunk C_i ($i \ge n{-}f$) is mapped to replica R_t, where $t = \psi(i, h) \bmod n$, using a public hash function ψ with block height h as salt, ensuring load diversity across blocks. At each height h, every replica performs full encoding but retains only the chunks assigned to it.

Hybrid Chunk Verification. To defend against malicious replicas returning tampered data, Hyra adopts a hybrid verification mechanism that combines *Merkle trees* and *Vector Commitments* (VCs) to ensure sub-chunk and chunk-level integrity, respectively.

Each chunk C_i is divided into s sub-chunks $\langle C_{i,0}, \ldots, C_{i,s-1} \rangle$. Hyra builds a Merkle tree $\mathcal{T}_i = \mathsf{Merkle}(C_i)$ over these sub-chunks, yielding a root digest θ_i. The root digests from all chunks collectively form a vector $V^h = \langle \theta_0, \ldots, \theta_{q-1} \rangle$, which is then committed using a VC scheme to produce $\Phi^h = \mathsf{Commit}(V^h)$. The target replica R_{target} storing chunk C_i computes and retains the VC commitment Φ^h and its opening proof $\Pi_i = \mathsf{Open}(V^h, \Phi^h, i)$ alongside the chunk.

When a sub-chunk $C_{i,j}$ is requested during read or recovery, R_{target} returns four parts: (i) the sub-chunk $C_{i,j}$; (ii) its Merkle proof $\pi_{i,j} = \mathsf{Prove}(\mathcal{T}_i, j)$; (iii) the root digest θ_i of $\mathcal{T}_i$; and (iv) the VC opening proof Π_i. Verification proceeds in two layers. First, $\mathsf{MVerify}(\theta_i, \pi_{i,j}, C_{i,j}, j)$ verifies that $C_{i,j}$ is correctly authenticated under θ_i. Then, $\mathsf{Verify}(\Phi^h, \Pi_i, \theta_i, i)$ confirms that θ_i is bound to index i in the committed vector V^h. To reduce storage overhead, the Merkle tree $\mathcal{T}_i$ is reconstructed based on C_i on demand rather than stored persistently.

This hybrid approach enables lightweight, fine-grained integrity checks at the sub-chunk level, while ensuring global consistency via vector commitments.

As an example, when sub-chunk $D_{0,0}$ is requested, the target replica returns: (i) the sub-chunk $D_{0,0}$; (ii) its Merkle proof $\pi_{0,0} = \langle (d_1, \mathrm{R}), (d_5, \mathrm{R}) \rangle$; (iii) the root digest $\theta_0 = d_6$ of the Merkle tree over

Algorithm 1: State Reading at Coordinator Replica

Input: $\langle \kappa, h \rangle$: state reading, H_{root}: hash of the root node at h, $\mathcal{M}_{root}$: metadata of the root node
Output: v: value of requested state κ

```
1  H_next ← H_root, M_next ← M_root, N_next ← null
2  while H_next ≠ null do
3      R_target ← target_Replica(M_root)
4      if R_target is current replica then
5          N_next ← load node with hash H_next
6      else
7          N_next ← remote_Node_Ret(R_target, M_next)
           // Timeout or incorrect node
8          if N_next = null or verify(N_next, H_next) is false then
               // Algorithm 2
9              N_next ← node_Recovery(H_next, M_next)
10     if N_next is a leaf node then
11         if match(N_next, κ) then
12             v ← state value in N_next
13             return v
14         else
15             return null
16     else
17         H_next, M_next ← next_Child(N_next, κ)
18 return null
```

C_0; and (iv) the VC opening proof Π_0 attesting that θ_0 is bound to index 0 in VC. Upon receiving this data, the verifying replica recomputes $d_0 = \mathsf{H}(D_{0,0})$, reconstructs the root via $\theta'_0 = \mathsf{H}\big(\mathsf{H}(d_0 \| d_1) \| d_5\big)$, and checks whether $\theta'_0 \stackrel{?}{=} \theta_0$. Next, the VC layer verifies that θ_0 is correctly bound to the commitment Φ^h at index 0 by invoking $\mathtt{Verify}(\Phi^h, \Pi_0, \theta_0, 0)$. This two-step verification succeeds only if the Merkle hashes are combined in the correct (position-dependent) order and if the digest θ_0 is authenticated under a VC.

Deferred and non-blocking encoding. Since $\mathfrak{N}_h$ is derived from the finalized block B_{h+1}, a replica may have already committed B_h (and thus become a member of $\mathfrak{N}_h$), but not yet received B_{h+1}. To bridge this gap, Hyra requires all potential members of $\mathfrak{N}_h$ to temporarily retain the original data chunks $\mathcal{D}^h$ before encoding and distribution take place. Once B_{h+1} is finalized and $\mathfrak{N}_h$ becomes locally agreed, as illustrated in Fig. 6, each replica in $\mathfrak{N}_h$ begins encoding $\mathcal{D}^h$ and distributing the resulting chunks. Since this process is decoupled from transaction execution, it runs asynchronously and off the critical path, introducing no blocking to the system's main workflow.

If all honest replicas in $\mathfrak{N}_h$ have completed encoding, Hyra can recover any missing data via decoding, as discussed earlier. Otherwise, any correct replica in $\mathfrak{N}_h$ that has not yet encoded still retains a full copy of the original data and can directly serve chunk requests. Therefore, replicas can flexibly perform encoding as background work, utilizing idle computational resources without interfering with core blockchain functionality.

4.4 State Reading

To retrieve a state value, the client issues a request $\langle \kappa, h \rangle$ to a designated coordinator replica R_c, where κ denotes the state key and h the target block height. The goal is to obtain the value of κ as of height h. Notably, the underlying partitioned storage is fully transparent to the client.

Algorithm 1 outlines the state reading procedure executed at R_c. It takes as input the request $\langle \kappa, h \rangle$, the root hash H_{root} of the state tree, and the root metadata $\mathcal{M}_{root}$, and returns the associated value v. The tree traversal proceeds top-down (Lines 2–17). For each intermediate node N_{next}, R_c determines its responsible replica R_{target}. If $R_{target} = R_c$, the node is retrieved locally (Lines 4–5); otherwise, a remote request is issued (Line 7). If the remote retrieval fails or returns invalid data, R_c initiates a recovery procedure via decoding, as detailed in Algorithm 2 (Lines 8–9). Once N_{next} is obtained, R_c checks whether it is a leaf node. If so, it compares the stored key suffix against κ. A match yields the corresponding value v (Lines 11–13); otherwise, the key is deemed absent (Line 15). If N_{next} is an internal node, traversal continues via the appropriate child pointer (Line 17). All nodes visited during the traversal collectively form a Merkle proof attesting to the correctness of the returned value. For clarity, proof construction is omitted from Algorithm 1.

Figure 7 illustrates the workflow of cross-block state reading. To serve a request $\langle \kappa, h \rangle$, Hyra traverses the state tree rooted at block height h, following key κ toward the corresponding leaf. At each step, the coordinator R_c fetches the next node using index metadata embedded in its parent. Assuming $R_c = R_0$, it retrieves N_0 locally, then uses metadata in N_0 to locate R_1 and fetch N_1, and so on, progressing through R_2, R_3, until reaching the leaf N_m. Once all nodes along the path are retrieved, the value for κ is returned. If a replica is unavailable or returns invalid data, R_c triggers decoding to recover the node. Note that Hyra partitions only historical states. Each replica maintains a local copy of the most recent state. During execution, all state accesses target only this up-to-date snapshot, which is locally available. As a result, execution remains entirely intra-replica and unaffected by the underlying partitioning strategy.

Lost Node Recovery. When a remote fetch fails, either due to timeout or failed verification, the coordinator R_c initiates recovery, as shown in Algorithm 2. Recovery proceeds incrementally from level 1 to L (Lines 2–8). At each level ℓ, R_c first locates the coding group G_ℓ that contains the missing node (Line 3) and identifies the set of replicas $\mathcal{R}$ responsible for storing the associated chunks (Line 4). R_c then broadcasts a request message msg to all replicas in $\mathcal{R}$ to solicit the required sub-chunks for decoding (Line 5) and waits for responses (Line 6). If enough valid sub-chunks are collected, decoding is performed at level ℓ; otherwise, the process escalates to the next level. Once the missing node is successfully reconstructed, it is returned to resume the ongoing state reading process (Lines 7–8).

Lines 18–23 describe how each replica R_k handles a recovery request $msg = \langle H_r, \mathcal{M}_r, \ell \rangle$ sent from coordinator R_c. The replica first determines the stripe range α, β that overlaps with the requested node (Line 19), then loads its local chunk C_{i_k} from the coding group G_ℓ. It extracts the relevant sub-chunks $C_{i_k,\alpha}, \ldots, C_{i_k,\beta}$ along with their Merkle proofs $\pi_{i_k,\alpha}, \ldots, \pi_{i_k,\beta}$, and generates the VC opening proof Π_{i_k} for the chunk root θ_{i_k} (Lines 20–21). These are returned to R_c as a response message.

Lines 9–17 describe how R_c processes these responses to perform decoding. Upon receiving a message msg_k from a replica, R_c first verifies the Merkle root θ_{i_k} using the VC opening proof Π_{i_k} against the local commitment Φ (Lines 10–11). Then, it verifies each sub-chunk using its Merkle path (Lines 12–14). Once K_ℓ valid sub-chunks from distinct replicas have been collected, R_c performs stripe-by-stripe decoding (Lines 15–17). After reconstructing the missing node N_r, the state reading process resumes.

Algorithm 2: Node Recovery

Input: H_r, $\mathcal{M}_r$: hash and metadata of the requested node
Output: N_r: requested node
/* **On coordinator replica** R_c */
$\ell \leftarrow 1$, $N_{next} \leftarrow$ null
for ℓ from 1 to L **do**
  $G_\ell \leftarrow$ `get_Coding_Group`$(\mathcal{M}_r, \ell)$
  $\mathcal{R} \leftarrow$ `get_Replicas`(G_ℓ)
  `broadcast`$(\mathcal{R}, msg = \langle H_r, \mathcal{M}_r, \ell \rangle)$
  `wait`()
  if $N_{next} \neq$ null **then**
    return N_r
Upon receive sub-chunks with proofs $msg_k = \langle \{C_{i_k,j}, \pi_{i_k,j} | j = \alpha, \ldots, \beta\}, \theta_{i_k}, \Pi_{i_k} \rangle$ **do**
  if Verify$(\Phi, \Pi_{i_k}, \theta_{i_k}, i_k)$ is false **then**
    return
  for j from α to β **do**
    if MVerify$(\theta_{i_k}, \pi_{i_k,j}, C_{i_k,j}, j)$ is false **then**
      return
  if R_c receive K_ℓ valid messages **then**
    $D_{r,\alpha}, \ldots, D_{r,\beta} \leftarrow$ `Decode`$(\{C_{i_1,j}\}, \ldots, \{C_{i_{K_\ell},j}\})$
    $N_r \leftarrow$ `extract_Node`$(D_{r,\alpha}, \ldots, D_{r,\beta}, \mathcal{M}_r)$
/* **On each other replica** R_k */
Upon receive request $msg = \langle H_r, \mathcal{M}_r, \ell \rangle$ from R_c **do**
  $\alpha, \beta \leftarrow$ `stripe_Range`$(\mathcal{M}_r.off, \mathcal{M}_r.len)$
  $G_\ell \leftarrow$ `get_Coding_Group`$(\mathcal{M}_r, \ell)$
  $\{C_{i_k,j}, \pi_{i_k,j} | j = \alpha, \ldots, \beta\}, \theta_{i_k}, \Pi_{i_k} \leftarrow$ `load_SubChunks`$(G_\ell, \mathcal{M}_r.h, \mathcal{M}_r.idx, \alpha, \beta)$
  $msg_k \leftarrow \langle \{C_{i_k,j}, \pi_{i_k,j} | j = \alpha, \ldots, \beta\}, \theta_{i_k}, \Pi_{i_k} \rangle$
  `send`(R_c, msg_k)

4.5 Locality-Aware Nodes Partitioning

Hyra partitions modified state tree nodes at each block height into data chunks. Naively hashing nodes for assignment ignores structural relationships, scattering related nodes across replicas and causing excessive cross-replica reads. Instead, Hyra uses a *locality-aware partitioning* scheme that leverages the tree's hierarchy and temporal access patterns.

Consider a read request $\langle \kappa, h \rangle$ where the latest update to κ occurred at height $h-2$. As illustrated in Fig. 7, the traversal spans multiple heights: root nodes at h, intermediates at $h-1$, and a leaf path at $h-2$. This reveals two locality patterns guiding Hyra's partitioning. (1) Root-near locality: Nodes near the root are frequently updated due to copy-on-write and shared across reads. Co-locating the root and its children in a chunk enables most traversals to fetch them in a single access, reducing redundant reads. (2) Leaf-level locality: Toward the leaves, the tree becomes sparse, often with single-child internal nodes. Updating a leaf also updates its ancestors, forming narrow linear paths at earlier heights. Historical reads must fetch the full path. Chunking these nodes together enables

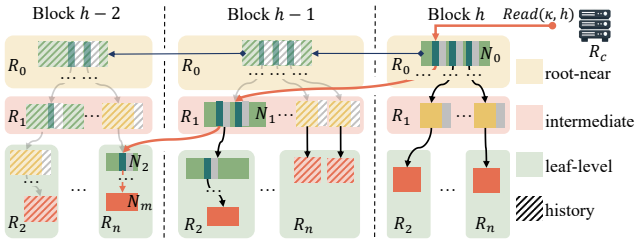


Fig. 7. Example of cross-block state reading

batch retrieval, reducing network round-trips. Based on these insights, Hyra groups updated nodes at each block height into three categories and applies tailored partitioning strategies to each.

- **Root-near nodes:** Small in number but highly reused across blocks. These are grouped into a single chunk, enabling the upper portion (typically the first 2–3 levels) of most read paths to be accessed in one round.
- **Leaf-level nodes:** These form narrow update paths and are grouped using *depth-first traversal*, which co-locates path-aligned nodes into the same chunk, optimizing for linear access patterns during historical reads.
- **Intermediate nodes:** These nodes are less predictable and shared across multiple paths, often with weaker locality. They are partitioned using a balanced *breadth-first traversal* to evenly distribute them across chunks.

In our evaluation with an MPT comprising 20K states distributed across 8 replicas, root-near nodes are typically drawn from levels 1–3, intermediate nodes from levels 3–4, and leaf-level nodes from levels 4–7. This hybrid partitioning strategy significantly enhances read locality by co-locating structurally and temporally related nodes, thereby reducing cross-replica communication and the need for decoding during reads.

5 Discussion and Analysis

This section proves the correctness of Hyra and analyzes the storage overhead caused.

5.1 Recoverability

In this subsection, we formally prove that the design of Hyra guarantees data recoverability, as stated in the following lemma.

Lemma 5.1. *For any data chunk D_i at block height h, Hyra ensures it is never permanently lost.*

Proof. We focus on the top-level encoding (level L), which guarantees recoverability. At each height h, the selected replica set $\mathfrak{N}_h$ contains $n-f$ replicas, among which at least $n-2f$ are honest. Under the $(n-2f, 2f)$ RS scheme, any $n-2f$ chunks suffice to reconstruct the original data chunks $\langle D_0, \ldots, D_{n-2f-1}\rangle$. $\mathfrak{N}_h$ is agreed upon via BFT consensus and recorded in the header of block B_{h+1}, ensuring all honest replicas share a consistent view. If an honest replica $R \in \mathfrak{N}_h$ has not yet performed encoding, it retains the full data $\mathcal{D}^h$ until learning that it is in $\mathfrak{N}_h$. Once the network becomes synchronous, R can either complete encoding or supply its data directly for recovery. By the eventual synchrony assumption, this ensures any D_i remains recoverable from honest replicas. □

We now show that the $(n-2f, f)$ coding configuration adopted by Hyra at the top level (L) is optimal in terms of redundancy. We define the *redundancy rate* as the ratio of the total number of stored chunks (data + parity) to the number of original data chunks. While lower-level parity chunks ($\ell < L$) are configurable for performance, the top-level encoding must guarantee recoverability under Byzantine conditions. Therefore, we aim to minimize the redundancy rate at level L subject to this availability constraint.

Lemma 5.2. *Under BFT environment with f Byzantine replicas, the redundancy rate of any coding scheme that guarantees recoverability must be at least $\frac{n-f}{n-2f}$.*

Proof. Let the top-level coding scheme use parameters (K_L, M_L), the redundancy rate is $d = K_L + M_L/K_L$. First, to ensure recoverability in the presence of f faulty replicas, at least K_L out of the $K_L + M_L$ chunks must be available. In the worst case, f replicas may withhold or return invalid chunks, so we must have:

$$K_L + M_L - f \geq K_L \Rightarrow M_L \geq f$$

Second, we consider the constraints imposed by BFT consensus. A block is committed when at least $n-f$ replicas agree. Since at most f of them may be faulty, it follows that at least $n-2f$ honest replicas have committed the previous block. Therefore, no more than $n-2f$ honest replicas are guaranteed to store data chunks at height h. To ensure successful recovery without relying on faulty replicas, the data must be recoverable from these $n-2f$ honest replicas. Thus, $K_L \leq n-2f$. Combining the above, we obtain the redundancy rate:

$$\frac{K_L + M_L}{K_L} \geq \frac{K_L + f}{K_L} \geq \frac{n-2f+f}{n-2f} = \frac{n-f}{n-2f}$$

This bound is tight and achieved when $K_L = n-2f$ and $M_L = f$, corresponding to the configuration used in Hyra. □

5.2 Storage Scalability

We analyze the average storage consumption at each block height in Hyra, and examine how the storage capacity scales with the number of replicas. Let S_h denote the total size of updated nodes at block height h. These nodes are partitioned into $K_L = n-2f$ data chunks, each of size S_h/K_L. The total number of chunks generated includes: K_L data chunks; $M_L = f$ top-level parity chunks; and $\sum_{\ell=1}^{L-1} \lceil K_L/K_\ell \rceil \cdot M_\ell$ parity chunks from lower levels. Hence, the total storage per block is:

$$\begin{aligned} S &= \left(K_L + M_L + \sum_{\ell=1}^{L-1} \left\lceil \frac{K_L}{K_\ell} \right\rceil \cdot M_\ell \right) \cdot \frac{S_h}{K_L} < \frac{n-f+\sum_{\ell=1}^{L-1} \left(\frac{n-2f}{K_\ell} + 1 \right) M_\ell}{n-2f} \cdot S_h \\ &= \left(1 + \frac{f}{n-2f} + \sum_{\ell=1}^{L-1} \left(\frac{M_\ell}{K_\ell} + \frac{M_\ell}{n-2f} \right) \right) S_h < \left(2 + \sum_{\ell=1}^{L-1} \left(\frac{M_\ell}{K_\ell} + M_\ell \right) \right) S_h = \mathcal{U} \end{aligned} \tag{2}$$

Here, (K_ℓ, M_ℓ) for $\ell < L$ are constants independent of n and f. Thus, the total storage per block height is bounded by a constant $\mathcal{U}$.

Then, we analyze the per-replica storage overhead $\frac{S}{n}$ at each height. From Eq. (2), we know that $S < \mathcal{U}$ is a constant independent of n. Therefore, $S/n < \mathcal{U}/n$, which decreases linearly as the number of replicas n increases. This implies: *the storage burden on each replica decreases with system scale, while the aggregate storage capacity of the system increases linearly with number of replicas.*

5.3 Reading Performance

State reading in Hyra involves traversing the state tree from the root to the target leaf node corresponding to the key, potentially accessing nodes stored across different replicas.

Node retrieval. When retrieving a node, there are two scenarios: (1) If the node is stored on an honest replica, the coordinator receives the correct response via a single network round, incurring minimal overhead, just the node and its corresponding Merkle/VC proof. (2) If the node is stored on a faulty replica (e.g., returns incorrect data or remains silent), the coordinator initiates decoding, starting from level 1 up to level L in the hierarchical coding scheme.

At level ℓ, the recovery process involves contacting up to $K_\ell + M_\ell$ replicas in the relevant coding group and requesting only the sub-chunks overlapping the target node. Let the node span $s = \lceil len/S_{\text{sub}} \rceil$ stripes, where len is the node size and S_{sub} is the sub-chunk size. Then, the communication overhead is bounded by $O(s \cdot K_\ell)$. If decoding fails at level ℓ, Hyra escalates to the next level with a larger K_ℓ. Assuming a fraction $\gamma = f/n$ of replicas are malicious, the probability of successful decoding at level ℓ is:

$$\mathbb{P}_\ell = \sum_{i=K_\ell}^{K_\ell+M_\ell} \binom{K_\ell + M_\ell}{i} (1-\gamma)^i \cdot \gamma^{K_\ell+M_\ell-i} \tag{3}$$

This represents the cumulative probability of receiving at least K_ℓ valid chunks. In most cases, decoding completes at level 1, keeping latency and communication low. To estimate the overall decoding cost, we define the expected number of contacted replicas as:

$$\overline{K} = \sum_{\ell=0}^{L} K_\ell \cdot \prod_{i=0}^{\ell-1} (1 - \mathbb{P}_i) \tag{4}$$

where we define $\mathbb{P}_0 = 1 - \gamma$ and $K_0 = 1$ to model direct access to an honest replica without decoding. According to the parameters used in our experiments, the $\overline{K}$ is between 3 to 5, significantly less than $n-2f$. Thus, the average communication complexity of node retrieval is $O(s \cdot \overline{K})$.

State reading. Hyra significantly enhances reading performance through its locality-aware partitioning. In hash-based partitioning, nodes are assigned to replicas based on the modulo of their hash values, causing logically related nodes, such as those along the same search path, to be randomly scattered. As a result, a single reading may involve multiple remote accesses, each retrieving only one relevant node. This low read coverage leads to high communication overhead and, in the worst case, incurs a reading complexity of $O(s \cdot \overline{K} \cdot \log L_r)$, where L_r is the length of the search path, s is the number of sub-chunks per node, and $\overline{K}$ is the average number of contacted replicas per node. In contrast, Hyra groups structurally and semantically related nodes into the same chunk using a hybrid partitioning strategy that considers both vertical (depth-wise) and horizontal (breadth-wise) locality. This co-location enables multiple nodes on the same query path to be fetched together, reducing the number of remote reads. As a result, the reading complexity becomes $O(\epsilon \cdot s \cdot \overline{K} \cdot \log L_r)$, where $0<\epsilon<1$ reflects the improved read coverage. Empirical results show that ϵ typically falls between 0.3 and 0.5.

6 Experiment

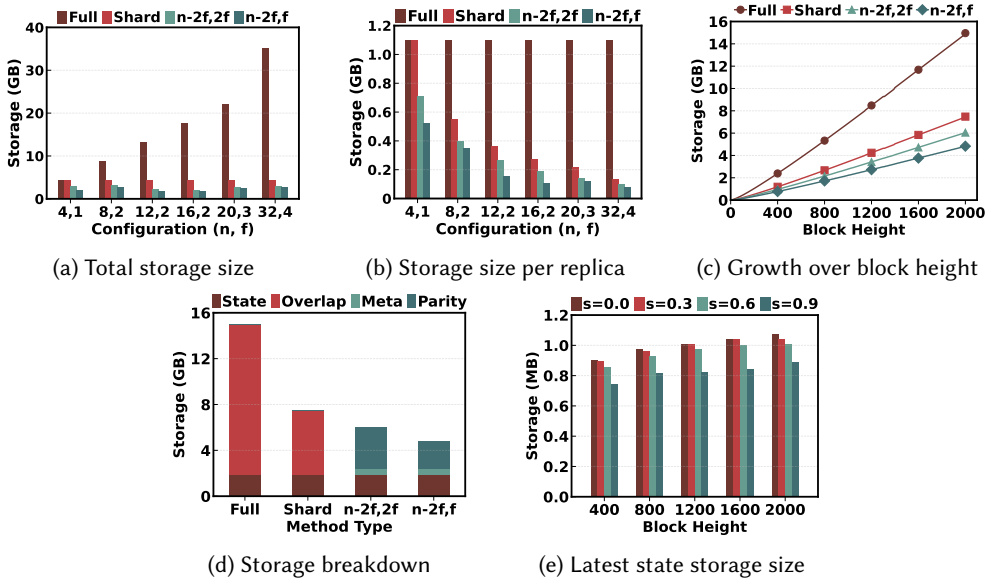
We evaluate the performance of Hyra in various settings. We describe the experimental setup.

6.1 Setup

Implementation. Hyra [4] is integrated into the official Ethereum client [1], augmented with an external PBFT module to provide consensus-layer finality. It uses Reed–Solomon erasure coding [57], SHA-256 as the hash function, and implements all cryptographic primitives, digital signatures, CoSi aggregation, and vector commitments, on the BN256 elliptic curve [50]. Replicas communicate via a full-mesh TCP overlay for peer-to-peer messaging. The state is maintained using the MPT from Section 2.3, consistent with account-based blockchains. The full prototype is implemented in C++ with 6,500 lines of code.

Comparisons. To assess the effectiveness of Hyra, we compare it with four representative baselines:

- *Full replication*: Every replica stores the entire state, ensuring zero access latency but incurring high storage and redundancy overhead.



(a) Total storage size (b) Storage size per replica (c) Growth over block height

(d) Storage breakdown (e) Latest state storage size

Fig. 8. Storage overhead under varying configuration.

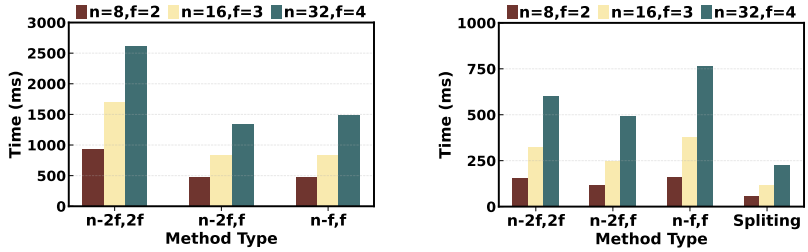
- *Sharding* [32, 35, 69]: The network is divided into multiple committees, each maintaining a disjoint portion of the global state and an independent ledger. Cross-shard operations require inter-committee coordination.
- *Flat encoding* [23, 39]: All state chunks are encoded as a single group using Reed-Solomon codes, without hierarchical structure. This provides fault tolerance but lacks decoding flexibility.
- *Hash-based partitioning* [9, 10, 68]: Each state node is assigned to a replica based on its hash value, either using simple modulo or consistent hashing in a circular identifier space (as in DHTs), where a node is placed on the first replica clockwise from its hash. These schemes are oblivious to the hierarchical structure and runtime access locality of nodes, often causing related nodes to be scattered across replicas, leading to high read latency.

In addition, we also evaluate Hyra under two different top-level coding schemes, $(n{-}2f, 2f)$ and $(n{-}2f, f)$, as discussed in Section 4.2.

Configuration. By default, Hyra uses a four-level hierarchical encoding scheme: $(K_1{=}2, M_1{=}1)$, $(K_2{=}4, M_2{=}2)$, $(K_3{=}8, M_3{=}3)$, and top-level $(K_4{=}n{-}2f, M_4{=}f)$. With $n = 32$ and $f = 10$, the top level is $(K_4 = 12,\ M_4 = 10)$, yielding a consistent hierarchy $(2, 1) \rightarrow (4, 2) \rightarrow (8, 3) \rightarrow (12, 10)$. For smaller K_4 (e.g., 4 or 8), fewer levels are used. For the flat encoding, we apply a single-level code directly, with parameters $(K_L{=}n{-}2f, M_L{=}f)$. With respect to chunk size, Hyra is evaluated with 200 KB, 500 KB, 1 MB, and 2 MB, while the sub-chunk size is fixed at 100 bytes. We also simulate Byzantine replicas, including data omission and incorrect responses.

Workloads. We use the SmallBank benchmark, widely adopted in prior work [19, 61, 66], to evaluate Hyra under a realistic, state-intensive workload. The dataset consists of one million unique accounts, each initialized with a random balance. We generate two million transfer transactions, where each transaction randomly selects two accounts and transfers funds between them, resulting in two reads and two writes per transaction. The access pattern follows a uniform distribution, applying consistent pressure across the entire state space. Unless otherwise specified, each block contains a fixed batch of 1,000 transactions.

Testbed. All experiments are conducted on a cluster of six machines within the same geographic region. Each machine is equipped with dual-socket Intel Xeon E5-2620 CPUs (2.10GHz, 4 cores per



(a) Encoding performance (b) Decoding performance

Fig. 9. Encoding and decoding performance

socket), 32GB RAM, 1TB disk, and a 1Gbps network interface. To evaluate scalability across different network sizes, we deploy up to two logical replicas per machine, a setup commonly adopted in prior work [27, 44]. All machines run Ubuntu 16.04 with g++ version 9.4. The number of replicas ranges from 4 to 128.

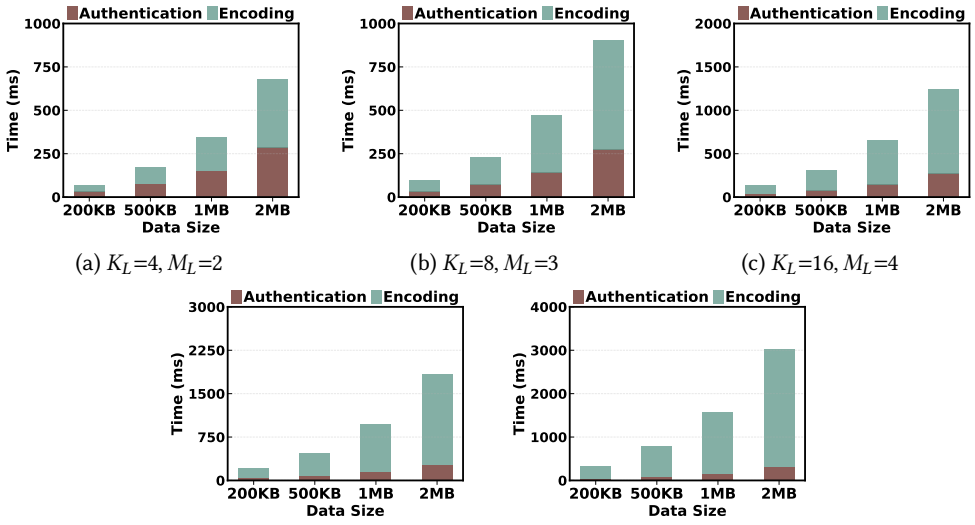
6.2 Storage Overhead

We evaluate the storage overhead of Hyra from two perspectives: the total storage consumption across all replicas and the average per-replica storage usage.

Overall storage scalability. Fig. 8a presents the total storage overhead over 1,200 consecutive blocks under different strategies and replica configurations. `Full` and `Shard` correspond to full replication and sharding (with 4 replicas per shard), respectively. Initially, all schemes incur similar overhead. However, `Full` grows linearly with the number of replicas, reaching 35.2 GB at $n = 32$, since each replica maintains a complete copy of the state. `Shard` reduces redundancy across shards but still replicates the state withing shard members, leading to moderate overhead. In contrast, Hyra scales much more efficiently: at $n = 32$, $f = 4$, it reduces the total overhead to just 2.6 GB. Fig. 8b shows the average per-replica storage cost. `Full` maintains a constant 1.10 GB per replica regardless of the number replicas, while `Shard` reduces this to 0.32 GB at $n = 32$. Hyra further lowers it to only 0.08 GB per replica, representing just 7.3% of full replication and 60.4% of sharding. Moreover, the $(n{-}2f, f)$ coding scheme incurs about 26.7% less storage overhead than $(n{-}2f, 2f)$ due to its reduced redundancy.

Under $n = 8$, $f = 2$ with a two-level hierarchical encoding scheme, Fig. 8c plots the storage overhead as block height increases. `Full` grows linearly, reaching 15 GB at height 2000, while `Shard` cuts this roughly in half (to about 8 GB). By contrast, Hyra significantly reduces overhead throughout, with the $(n{-}2f, f)$ configuration achieving the lowest storage and up to 66% reduction compared to full replication. These results confirm that Hyra's partitioning strategy substantially improves blockchain storage scalability.

Storage breakdown. Fig. 8d breaks down the storage overhead into four components: state, replication, metadata, and parity across different methods (evaluated under $n = 8$, $f = 2$ with a two-level hierarchical encoding scheme). In `Full`, redundancy is high because each replica stores the complete state, resulting in a total storage of 15 GB. `Shard` reduces this overhead by distributing state data across multiple shards, yet each shard still retains full replication within each shard. In contrast, Hyra, under both schemes, eliminates replication and introduces only moderate metadata and parity overhead. Notably, the overhead of these in the $(n{-}2f, f)$ scheme is only 32.4% of the replication cost in `Full`. Fig. 8e evaluates the storage cost of maintaining the latest state snapshot across different block heights under varying skewness factors s. State access follows a Zipfian distribution [8], where a higher s indicates stronger skewness, and s=0 corresponds to uniform access. The latest state size remains consistently small (around 0.9–1.1 MB), even as s increases



(a) K_L=4, M_L=2 (b) K_L=8, M_L=3 (c) K_L=16, M_L=4

(d) K_L=32, M_L=5 (e) K_L=64, M_L=6

Fig. 10. Overhead of hierarchical encoding and authentication.

from 0.0 to 0.9. This is because the size depends primarily on the total number of states, not the distribution of access frequencies. Thus, maintaining a full, up-to-date state locally at each replica is both practical and lightweight.

6.3 Encoding and Decoding Efficiency

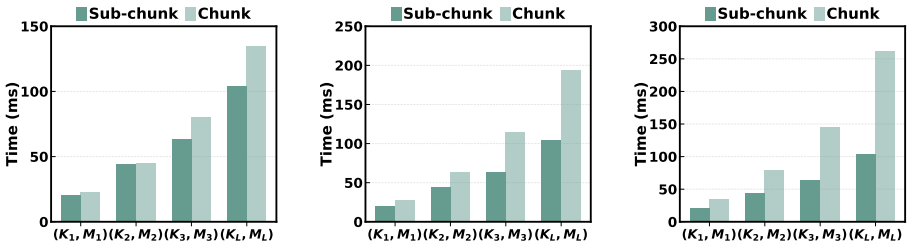
Next, we evaluate the performance of Hyra's encoding and decoding mechanisms under different coding configurations.

Coding scheme. Fig. 9 presents encoding and decoding performance across several coding schemes without hierarchical encoding, to isolate the impact of top-level configurations. We compare three schemes: $(n{-}2f, f)$ (adopted by Hyra), $(n{-}2f, 2f)$ (used by BFT-Store), and $(n{-}f, f)$ (which lacks fault tolerance but serves as a performance baseline). For decoding, we additionally include the `Splitting` strategy, which enables sub-chunk-level decoding as described in Section 4.

Fig. 9a reports encoding time under each scheme. As n and f increase, encoding time grows due to the larger number of parity chunks generated. Among all, the $(n{-}2f, f)$ scheme used by Hyra achieves the best efficiency due to its lower redundancy overhead. In contrast, $(n{-}2f, 2f)$, with twice the parity, incurs the highest cost, reaching 2608.0 ms at n=32, f=4. The $(n{-}f, f)$ and Hyra 's $(n{-}2f, f)$ configurations show comparable and significantly lower encoding times (e.g., 468.5 ms at n=8, f=2), confirming the efficiency of the scheme chosen by Hyra.

Fig. 9b shows the corresponding decoding times. As expected, decoding slows with increasing n and f, but clear differences emerge. The $(n{-}f, f)$ scheme performs the worst, requiring nearly all chunks for recovery, with latency peaking at 764.0 ms for n=32, f=4. Both $(n{-}2f, 2f)$ and $(n{-}2f, f)$ schemes achieve lower latency, with Hyra's configuration slightly outperforming the other. Notably, the `Splitting` approach delivers the best performance (224.0 ms at n=32, f=4), as it fetches and reconstructs only the necessary sub-chunks, reducing both bandwidth and computation costs.

Hierarchical encoding. Fig. 10 presents the time overhead of hierarchical encoding under different top-level parameters ($K_L = n{-}2f, M_L = f$), while fixing the parameters of lower levels. The reported latency includes both *encoding* and *authentication*, where authentication encompasses the construction of VC commitments and Merkle trees over sub-chunks. The results show that total construction time grows linearly with data size, as larger inputs involve more computation



(a) 200KB (b) 1MB (c) 2MB

Fig. 11. Efficiency of hierarchical decoding

and memory operations. Moreover, increasing the coding parameters (K_L, M_L) results in higher encoding times due to increased redundancy and more parity chunks. For instance, encoding 2 MB of data under $(4, 2)$ takes 682.6 ms, while $(8, 3)$ and $(16, 4)$ increase this to 905.0 ms and 1248.0 ms, respectively. Despite these trends, the overhead of authentication remains relatively small, typically under 40% of the total encoding time, and sometimes as low as 5%, indicating that most of the cost stems from erasure coding rather than proof construction.

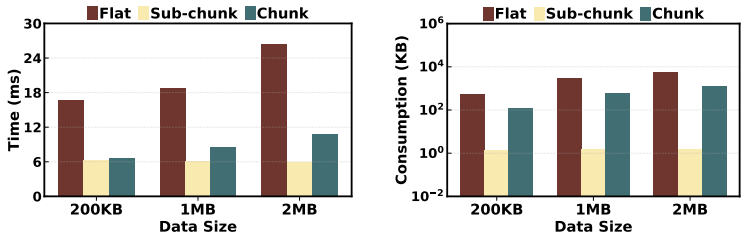
Hierarchical decoding. Fig. 11 evaluates the decoding performance at each individual level under fixed parameters $n = 32$ and $f = 10$, with varying chunk sizes. The coding scheme at each level follows the configuration described earlier. This experiment focuses purely on decoding time, excluding network latency and request overhead. We compare two decoding strategies: `Chunk`, which performs decoding over the entire chunk, and `Sub-chunk`, which decodes only the overlapping sub-chunks necessary to recover the requested node. As expected, decoding time increases with the level, since higher levels correspond to larger coding groups. For example, the decoding time grows from 25.1 ms at level 1 to 190.0 ms at the top level ($L = 4$). Notably, `Sub-chunk` consistently outperforms `Chunk`, especially at higher levels. Specifically, `Sub-chunk` decoding time remains around 100.3 ms at level L regardless of chunk size, as it processes only the relevant sub-chunks instead of entire chunks, thereby reducing unnecessary computation.

6.4 Node Retrieval Efficiency

Third, we test Hyra when retrieving a state node, including remote data access, integrity verification, and decoding.

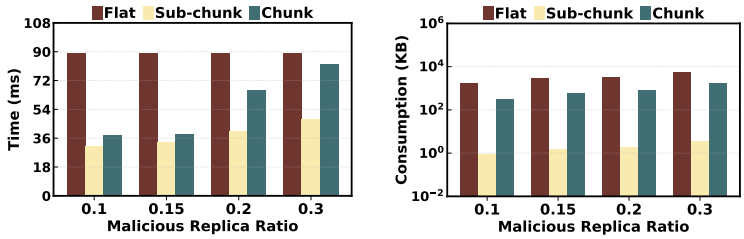
Node retrieval. Fig. 12a reports the latency of retrieving a single node under three strategies: `Flat`, `Chunk`, and `Sub-chunk`, with fixed parameters $n = 32$ and $f = 10$, and varying chunk sizes. Here, `Flat` refers to the baseline that applies erasure coding only at the top level without hierarchical structure. The results show that both `Chunk` and `Sub-chunk` significantly outperform `Flat`, as in over 80% of cases, decoding is either unnecessary or completed at level 1. Furthermore, `Sub-chunk` reduces latency by up to 55.8% compared to `Chunk`, due to its fine-grained data access and reduced verification overhead. Fig. 12b presents the corresponding bandwidth cost for each strategy. As expected, `Flat` incurs the highest overhead since it often retrieves entire chunks or full coding groups. `Chunk` improves over `Flat` by leveraging hierarchical decoding, but still transmits full chunks. In contrast, `Sub-chunk` achieves the lowest bandwidth usage, transferring only 1.1% of the data required by `Chunk` and 0.23% of that required by `Flat`.

Malicious attacks. Fig. 13 evaluates the performance of Hyra under different Byzantine fault ratios (f/n), where faulty replicas either remain silent or return corrupted data. The chunk size is fixed at 1 MB. Fig. 13a shows the node request latency as the fraction of malicious replicas increases. As expected, the latency of all methods grows moderately due to increased decoding attempts. For instance, the latency of `Chunk` increases from 38.3 ms at 10% faults to 82.2 ms at



(a) Retrieval performance (b) Bandwidth consumption

Fig. 12. Retrieval efficiency under varying data size



(a) Retrieval performance (b) Bandwidth consumption

Fig. 13. Retrieval efficiency under varying malicious ratio

30% faults, still significantly lower than the constant 90.1 ms required by `Flat`, which always decodes at the top level. Fig. 13b reports the corresponding bandwidth consumption. For `Chunk`, bandwidth steadily increases with more faults, reaching nearly 1,700 KB at a 30% malicious ratio due to repeated full-chunk recovery. In contrast, `Sub-chunk` maintains exceptionally low bandwidth in all scenarios, staying below 10 KB even under high Byzantine presence. These results demonstrate that `Sub-chunk` reduces communication overhead, making it a highly bandwidth-efficient strategy.

6.5 Reading Performance

Finally, we evaluate the state reading performance of Hyra. To isolate the impact of cross-replica access, the benchmark focuses solely on historical state, excluding the most recent locally stored state. Each read targets a single key from the historical state.

Fig. 14 compares the average number of remote accesses per read under three partitioning strategies: `Random`, `DHT`, and our proposed `Hyra`. Both `Random` and `DHT` assign state nodes purely based on their hashes and are oblivious to structural relationships, as discussed earlier. For fair comparison, each chunk is counted at most once per read, even if it contains multiple requested nodes. The x-axis denotes the number of replicas, and the y-axis shows the average number of remote accesses over 10,000 queries. As the number of replicas increases, `Random` and `DHT` exhibit similarly high cross-replica access rates (e.g., from 4.31 at n=8 to 5.02 at n=128), reflecting poor spatial locality. In contrast, `Hyra` maintains low remote access counts (1.8–2.4), achieving a 30–60% reduction in cross-node reads.

Table 1 presents both average and maximum state reading latencies across varying replica counts. `Random` and `DHT` consistently exhibit higher latencies, increasing from 76 ms at $n = 4$ to 102 ms at $n = 128$, with worst-case delays reaching 142 ms. In comparison, `Hyra` achieves significantly lower and more stable latencies, with average latency reduced to 46 ms at $n = 128$. These results confirm that only `Hyra` effectively reduces cross-replica lookups and mitigates long-tail delays. For reference, the `Full` and `Sharded` strategies, denoted collectively as `Local`, achieve near-zero reading latency, averaging 73 µs and peaking at 96 µs, due to purely local access without any remote communication. While these approaches offer optimal read performance, they sacrifice scalability

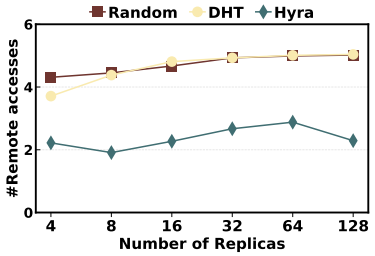


Fig. 14. Remote access

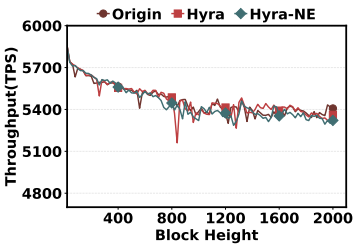


Fig. 15. System throughput

Table 1. Read latency under different partition methods

r	Avg. latency (ms)				Max. latency (ms)			
	Random	DHT	**Hyra**	Local	Random	DHT	**Hyra**	Local
4	76	75	45	0.073	142	142	60	0.096
8	90	88	39	0.073	142	142	60	0.096
16	96	97	46	0.073	141	141	81	0.096
32	100	100	54	0.073	142	142	81	0.096
64	101	102	58	0.073	142	142	81	0.096
128	101	102	46	0.073	142	142	61	0.096

and fault tolerance, underscoring Hyra's goal of enabling decentralized, resilient state storage with low read overhead.

Fig. 15 reports throughput across block heights for three configurations: the baseline `Origin` (without state partitioning), `Hyra`, and `Hyra-NE` (which decouples agreement on $\mathfrak{N}_h$ from consensus). The nearly overlapping curves confirm that Hyra's asynchronous, hierarchical encoding does not interfere with transaction execution. In addition, embedding the $\mathfrak{N}_h$ agreement into the consensus phase introduces minimal communication overhead, with no noticeable impact on throughput. The slight decline in performance at higher block heights stems from the larger MPT structure, which increases the number of nodes updated per block.

7 Related Works

Sharding. Although sharding is not designed to relieve the single-node storage cost, it can still achieve data partition across nodes "equivalently" [10, 35, 64]. Specifically, nodes are divided into multiple equal-size groups in blockchain sharding, called *shards*, while the states are distributed over these shards. Then, each shard only handles transactions that access states stored by itself in parallel to maximize the throughput. In such a scenario, each node only needs to preserve the state data assigned to the shard it belongs to instead of all the global state data. When encountering a transaction that accesses states from two different shards, various cross-shard commit protocols [54, 64, 69] are proposed to commit such kind of transactions. In fact, the sharding technique is orthogonal to our approaches. As each shard in a sharding system maintains a local chain independently, we can further diminish the storage overhead by employing our Hyra for state data management, incurring no modification to the sharding design.

Authenticated storage systems. Merkle tree [46] and its variants [11, 15, 65] widely serve as the underlying authenticated storage to manage the states of smart contracts. For a state request, in addition to the state value, Merkle tree can produce corresponding Merkle proof to authenticate the correctness of the value. Specifically, the proof includes the nodes visited in this query, from the root to the leaf node, that accommodate the target state. However, with more states inserted, the size of the proof rises accordingly, mainly since each node contains the hashes of all its children.

To pursue succinct proofs, more studies have changed to leverage various accumulators [14] or cryptographic structures to compress the proof size while not sacrificing security. The RSA-based accumulators can generate a constant proof size for each state regardless of the number of states. The *vector commitment* (VC) scheme [17] is another typical accumulator, providing the authentication feature. Differently, it accepts an array of data as input and generates a constant-size proof for data at each slot regardless of the array length. Based on VC, Edrax [20] significantly reduces the space overhead of proofs. Besides, VC is employed in stateless cryptocurrency design [22, 58]. In this case, the consensus node need not maintain entire state data; users store their state data and VC proofs instead. However, these solutions cannot support the execution of smart contract transactions and suffer from the time-consuming proof generation. Pointproofs [28] provides a more powerful VC scheme, which enables smart contract execution under a stateless design. Unfortunately, this requires storing a large number of parameters on each node, which is impractical in real deployments.

Storage partition. In traditional distributed systems, partitioning is an effective strategy for reducing per-node storage overhead by dividing data into chunks and distributing them across nodes [12, 18, 31, 36, 59]. To ensure availability, each chunk is typically replicated across multiple nodes. However, this approach is inadequate in Byzantine environments, where malicious replicas may tamper with data or respond dishonestly. To tolerate f Byzantine faults, each partition must be replicated to more than $n - f$ nodes, and authenticated retrieval becomes necessary, resulting in $O(n)$ storage complexity. To address this, recent blockchain systems adopt *erasure coding* (EC) to reduce redundancy while preserving recoverability. For example, BFT-Store [53, 55, 56] applies EC to block data, enabling secure partitioning under BFT assumptions. However, its design targets flat, append-only ledger data and does not generalize to the structurally dependent and frequently updated state data. In contrast, our work focuses on partitioning and encoding the internal nodes of the state tree, which presents unique challenges in verification, recoverability, and query efficiency.

8 Conclusion

In this paper, we presented Hyra, a Byzantine-resilient and storage-efficient state management engine for blockchain systems. Hyra combines locality-aware partitioning with hierarchical erasure coding to achieve scalable and fault-tolerant storage, while maintaining efficient access to distributed state. To ensure correctness under adversarial conditions, it integrates a hybrid verification mechanism using vector commitments and Merkle proofs for sub-chunk-level validation. Our formal analysis proves that the coding configuration is redundancy-optimal. Experiments on Ethereum-based prototype show that Hyra reduces per-replica storage by over 90% and achieves low-latency, verifiable access even in BFT environment. These results highlight Hyra as a practical and effective solution for secure, scalable state storage in future blockchain systems.

Acknowledgments

This work was supported by the National Key Research and Development Program of China under Grant 2023YFC3304700, the Shanghai Academic/Technology Research Leader Program under Grant 23XD1401100, and the Nanyang Technological University Centre for Computational Technologies in Finance (NTU-CCTF). The views expressed are those of the authors and do not necessarily reflect those of NTU-CCTF.

References

[1] 2019. C++ Ethereum. https://github.com/ethereum/aleth.
[2] 2025. Cosmos. https://cosmos.network/.
[3] 2025. Diem. https://https://github.com/diem/.
[4] 2025. Hyra. https://github.com/qqf0312/hyra.
[5] 2025. Optimism. https://community.optimism.io/docs/.
[6] 2025. Polygon. https://polygon.technology.
[7] 2025. Solana. https://solana.com/solana-whitepaper.pdf.
[8] 2025. Zipf's law. https://en.wikipedia.org/wiki/Zipf%27s_law.
[9] Ryosuke Abe, Shigeya Suzuki, and Jun Murai. 2018. Mitigating bitcoin node storage size by DHT. In *Proceedings of the 14th Asian Internet Engineering Conference*. 17–23.
[10] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. 2017. Chainspace: A sharded smart contracts platform. *arXiv preprint arXiv:1708.03778* (2017).
[11] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*. 1–15.
[12] David F Bacon, Nathan Bales, Nico Bruno, Brian F Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, et al. 2017. Spanner: Becoming a SQL system. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 331–343.
[13] Arati Baliga, I Subhod, Pandurang Kamat, and Siddhartha Chatterjee. 2018. Performance evaluation of the quorum blockchain platform. *arXiv preprint arXiv:1809.03421* (2018).
[14] Josh Benaloh and Michael De Mare. 1993. One-way accumulators: A decentralized alternative to digital signatures. In *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer, 274–285.
[15] Ethan Buchman. 2016. *Tendermint: Byzantine fault tolerance in the age of blockchains*. Ph. D. Dissertation. University of Guelph.
[16] Miguel Castro, Barbara Liskov, et al. 1999. Practical byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.
[17] Dario Catalano and Dario Fiore. 2013. Vector commitments and their applications. In *International Workshop on Public Key Cryptography*. Springer, 55–72.
[18] Stefano Ceri, Barbara Pernici, and Gio Wiederhold. 1987. Distributed database design methodologies. *Proc. IEEE* 75, 5 (1987), 533–546.
[19] Zhihao Chen, Tianji Yang, Yixiao Zheng, Zhao Zhang, Cheqing Jin, and Aoying Zhou. 2024. Spectrum: Speedy and Strictly-Deterministic Smart Contract Transactions for Blockchain Ledgers. *Proceedings of the VLDB Endowment* 17, 10 (2024), 2541–2554.
[20] Alexander Chepurnoy, Charalampos Papamanthou, Shravan Srinivasan, and Yupeng Zhang. 2018. Edrax: A cryptocurrency with stateless transaction validation. *Cryptology ePrint Archive* (2018).
[21] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.
[22] Thaddeus Dryja. 2019. Utreexo: A dynamic hash-based accumulator optimized for the Bitcoin UTXO set. *IACR Cryptol. ePrint Arch.* 2019 (2019), 611.
[23] Zhengyi Du, Xiongtao Pang, and Haifeng Qian. 2021. Partitionchain: A scalable and reliable data storage strategy for permissioned blockchain. *IEEE Transactions on Knowledge and Data Engineering* 35, 4 (2021), 4124–4136.
[24] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* 35, 2 (1988), 288–323.
[25] Michael Eischer and Tobias Distler. 2019. Scalable Byzantine fault-tolerant state-machine replication on heterogeneous servers. *Computing* 101, 2 (2019), 97–118.
[26] Robert Escriva, Bernard Wong, and Emin Gün Sirer. 2012. HyperDex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. 25–36.
[27] Yossi Gilad, Rotem Hemo, Silvio Micali, et al. 2017. Algorand: Scaling byzantine agreements for cryptocurrencies. In *SOSP*. ACM, 51–68.
[28] Sergey Gorbunov, Leonid Reyzin, Hoeteck Wee, and Zhenfei Zhang. 2020. Pointproofs: aggregating proofs for multiple vector commitments. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2007–2023.
[29] Zicong Hong, Song Guo, Enyuan Zhou, Jianting Zhang, Wuhui Chen, Jinwen Liang, Jie Zhang, and Albert Zomaya. 2023. Prophet: Conflict-free sharding blockchain via byzantine-tolerant deterministic ordering. In *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*. IEEE, 1–10.

[30] Ling Hu, Wei-Shinn Ku, Spiridon Bakiras, and Cyrus Shahabi. 2013. Spatial Query Integrity with Voronoi Neighbors. *IEEE Transactions on Knowledge and Data Engineering* 25 (2013), 863–876. https://api.semanticscholar.org/CorpusID:1340246

[31] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[32] Huawei Huang, Xiaowen Peng, Jianzhou Zhan, Shenyang Zhang, Yue Lin, Zibin Zheng, and Song Guo. 2022. Brokerchain: A cross-shard blockchain protocol for account/balance-based state sharding. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE, 1968–1977.

[33] Cheqing Jin, Shuaifeng Pang, Xiaodong Qi, Zhao Zhang, and Aoying Zhou. 2021. A high performance concurrency protocol for smart contracts of permissioned blockchain. *IEEE Transactions on Knowledge and Data Engineering* 34, 11 (2021), 5070–5083.

[34] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S Matthew Weinberg, and Edward W Felten. 2018. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*. 1353–1370.

[35] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE symposium on security and privacy (SP)*. IEEE, 583–598.

[36] Chuck Lam. 2010. *Hadoop in action*. Simon and Schuster.

[37] Bijun Li, Wenbo Xu, Muhammad Zeeshan Abid, Tobias Distler, and Rüdiger Kapitza. 2016. Sarek: Optimistic parallel ordering in byzantine fault tolerance. In *2016 12th European Dependable Computing Conference (EDCC)*. IEEE, 77–88.

[38] Chenxin Li, Peilun Li, Dong Zhou, Zhe Yang, Ming Wu, Guang Yang, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. 2020. A decentralized blockchain with high throughput and fast confirmation. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 515–528.

[39] Chunlin Li, Jing Zhang, Xianmin Yang, and Luo Youlong. 2021. Lightweight blockchain consensus mechanism and storage optimization for resource-constrained IoT devices. *Information Processing & Management* 58, 4 (2021), 102602.

[40] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. 2006. Dynamic authenticated index structures for outsourced databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 121–132.

[41] Benoît Libert and Moti Yung. 2010. Concise mercurial vector commitments and independent zero-knowledge sets with short proofs. In *Theory of Cryptography Conference*. Springer, 499–517.

[42] Shengyun Liu, Wenbo Xu, Chen Shan, Xiaofeng Yan, Tianjing Xu, Bo Wang, Lei Fan, Fuxi Deng, Ying Yan, and Hui Zhang. 2023. Flexible advancement in asynchronous bft consensus. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 264–280.

[43] Lingling Lu, Zhenyu Wen, Ye Yuan, Qinming He, Jianhai Chen, and Zhenguang Liu. 2024. ANNProof: Building a verifiable and efficient outsourced approximate nearest neighbor search system on blockchain. *Future Generation Computer Systems* 156 (2024), 206–220. doi:10.1016/j.future.2024.03.002

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

[45] Ralph C Merkle. 1982. Method of providing digital signatures. US Patent 4,309,569.

[46] Ralph C Merkle. 1989. A certified digital signature. In *Conference on the Theory and Application of Cryptology*. Springer, 218–238.

[47] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The honey badger of BFT protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 31–42.

[48] Manas Minglani, Jim Diehl, Xiang Cao, Binghze Li, Dongchul Park, David J Lilja, and David HC Du. 2017. Kinetic action: Performance analysis of integrated key-value storage devices vs. leveldb servers. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 501–510.

[49] Ke Mu, Bo Yin, Alia Asheralieva, and Xuetao Wei. 2024. Separation is good: A faster order-fairness Byzantine consensus. (2024).

[50] Michael Naehrig, Ruben Niederhagen, and Peter Schwabe. 2010. New software speed records for cryptographic pairings. In *International Conference on Cryptology and Information Security in Latin America*. Springer, 109–123.

[51] Satoshi Nakamoto et al. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

[52] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[53] Xiaodong Qi, Zhihao Chen, Zhao Zhang, Cheqing Jin, Aoying Zhou, Haizhen Zhuo, and Quanqing Xu. 2021. A Byzantine Fault Tolerant Storage for Permissioned Blockchain. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*. 2673–2676.

[54] Xiaodong Qi, Zhihao Chen, Haizhen Zhuo, Quanqing Xu, Chengyu Zhu, Zhao Zhang, Cheqing Jin, Aoying Zhou, Ying Yan, and Hui Zhang. 2023. SChain: Scalable Concurrency over Flexible Permissioned Blockchain. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 1901–1913.

[55] Xiaodong Qi, Zhao Zhang, Cheqing Jin, and Aoying Zhou. 2020. BFT-Store: Storage partition for permissioned blockchain via erasure coding. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1926–1929.
[56] Xiaodong Qi, Zhao Zhang, Cheqing Jin, and Aoying Zhou. 2020. A Reliable Storage Partition for Permissioned Blockchain. *IEEE Transactions on Knowledge and Data Engineering* 33, 1 (2020), 14–27.
[57] I. S. Reed and G. Solomon. 1960. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial Applied Mathematics* (1960).
[58] Leonid Reyzin, Dmitry Meshkov, Alexander Chepurnoy, and Sasha Ivanov. 2017. Improving authenticated dynamic dictionaries, with applications to cryptocurrencies. In *International Conference on Financial Cryptography and Data Security*. Springer, 376–392.
[59] Salman Salloum, Ruslan Dautov, Xiaojun Chen, Patrick Xiaogang Peng, and Joshua Zhexue Huang. 2016. Big data analytics on Apache Spark. *International Journal of Data Science and Analytics* 1 (2016), 145–164.
[60] A Secure. 2018. The zilliqa project: A secure, scalable blockchain platform. (2018).
[61] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. 2021. Basil: Breaking up BFT with ACID (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 1–17.
[62] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, and Bryan Ford. 2016. Keeping authorities" honest or bust" with decentralized witness cosigning. In *2016 IEEE Symposium on Security and Privacy (SP)*. Ieee, 526–545.
[63] Michalis Vardoulakis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2022. Tebis: index shipping for efficient replication in lsm key-value stores. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 85–98.
[64] Jiaping Wang and Hao Wang. 2019. Monoxide: Scale out blockchains with asynchronous consensus zones. In *16th USENIX symposium on networked systems design and implementation (NSDI 19)*. 95–112.
[65] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.
[66] Cheng Xu, Ce Zhang, Jianliang Xu, and Jian Pei. 2021. SlimChain: Scaling blockchain transactions through off-chain storage and parallel processing. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2314–2326.
[67] Minghui Xu, Hechuan Guo, Ye Cheng, Chunchi Liu, Dongxiao Yu, and Xiuzhen Cheng. 2025. EC-Chain: Cost-Effective Storage Solution for Permissionless Blockchains. In *IEEE INFOCOM 2025-IEEE Conference on Computer Communications*. IEEE, 1–10.
[68] Zihuan Xu, Siyuan Han, and Lei Chen. 2018. CUB, a consensus unit-based storage scheme for blockchain system. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 173–184.
[69] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. 2018. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 931–948.
[70] Gengrui Zhang, Fei Pan, Sofia Tijanic, and Hans-Arno Jacobsen. 2024. Prestigebft: Revolutionizing view changes in bft consensus algorithms with reputation mechanisms. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 1930–1943.

Received July 2025; revised October 2025; accepted November 2025