MMR-Based Sequencing Commitment for Kaspa: Unifying Transaction Ordering and Logarithmic Inclusion Proofs

Arthur Zhang[*]
Tondi Foundation

Neo Maxwell
Tondi Foundation

January 2026

Abstract

We propose a Merkle Mountain Range (MMR) based Sequencing Commitment scheme for the Kaspa blockDAG, which provides a cryptographic commitment to the global canonical order of transactions while enabling compact $\mathcal{O}(\log n)$ inclusion proofs. This unified approach achieves the goals of Kaspa Improvement Proposals KIP-15 (canonical transaction ordering commitment) and KIP-6 (logarithmic receipt proofs) in a single design. We detail the modifications to Kaspa's consensus (new block header fields and validation rules), the MMR data structure integration for an append-only history of accepted transactions, and the generation of compact proofs that a given transaction was accepted at a certain position in the history.

We analyze the overhead of the scheme and demonstrate that it adds negligible bandwidth ($< 0.1\%$) and moderate storage requirements (tens of GB/year at 10 blocks per second)—a cost well justified by the benefits. The MMR commitments enable trustless light-client protocols and cross-chain bridges: for example, verifying a transaction from 100,000 blocks ago requires only about 17 hashes (proof under 1 KB) in this scheme, versus 100,000 sequential hashes in a traditional chain commitment. We present use cases such as trustless bridging to Ethereum-style contracts, fast bootstrapping of new nodes via skip proofs, and robust disaster recovery using verifiable checkpoints.

The scheme turns the entire blockchain history into a *self-authenticating log*: any single block's wrong data breaks the MMR root instantly, providing cryptographic guarantees that the whole history is verifiable against the latest block. This is analogous to how a Merkle root turns a block's transactions into a verifiable set, now extended to the entire blockchain level. Finally, we compare the MMR-based approach to the original KIP-15/KIP-6 proposals, showing that our unified design is more efficient and easier to implement consistently, with gas costs for on-chain verification reduced by 99.98% for transactions 100,000 blocks deep.

Keywords: Merkle Mountain Range, BlockDAG, Transaction Ordering, Inclusion Proofs, Layer-2 Scaling, Kaspa, GHOSTDAG, Light Client, Cross-Chain Bridge

[*]Corresponding author: Arthur Zhang (arthur@tondi.org)

Contents

1 Introduction

Blockchain systems must provide verifiable assurances of transaction inclusion and execution order for clients and Layer-2 protocols. In traditional single-chain blockchains (e.g., Bitcoin or Ethereum), a linear chain of blocks naturally establishes a total order of transactions, and Merkle trees within each block provide proofs of inclusion. A light client can verify a transaction by checking a Merkle proof against a known block header and ensuring that the block is in the canonical chain (typically by verifying a chain of block hashes). However, this approach has limitations when verifying transactions deep in history or when the blockchain structure is not a simple chain.

A well-defined global order of transactions is crucial for Layer-2 (L2) systems such as rollups and cross-chain bridges, which rely on the main chain (Layer-1) to provide an ordered log of transactions for verification and dispute resolution. In a linear blockchain, transaction order is simply the block sequence; but in a DAG-based protocol, multiple blocks can be mined concurrently, so one must extract a single "official" sequence of transactions from the partial order of blocks. This challenge has motivated the development of succinct proof protocols such as FlyClient [11] and NiPoPoWs (Non-Interactive Proofs of Proof-of-Work) [12], which use block sampling and interlinking to achieve logarithmic proof sizes. However, these protocols focus primarily on proving chain connectivity, not necessarily the inclusion and ordering of specific transactions within a DAG structure.

Kaspa is a high-throughput blockDAG (Directed Acyclic Graph) protocol where blocks are produced in parallel (multi-leader consensus). Rather than a single chain of blocks, Kaspa's consensus (the GHOSTDAG protocol) orders blocks in a DAG, identifying a *selected parent chain* (analogous to a backbone chain) and merging other parallel blocks (called the mergeset) into the ledger. In such a system, determining a global transaction order is non-trivial: multiple blocks (produced at nearly the same time) may contain different transactions, and the protocol must decide a consistent inclusion order for all transactions.

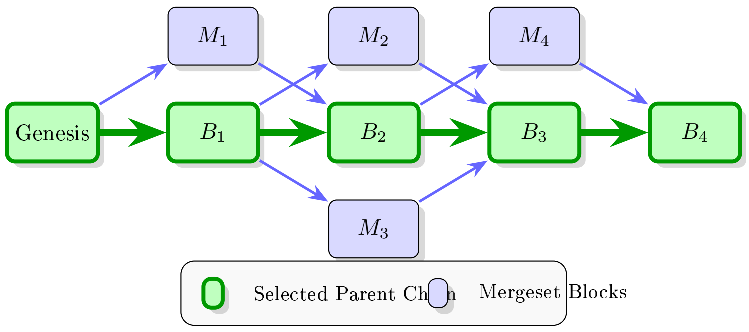


Figure 1: Kaspa BlockDAG Structure. The green blocks form the *selected parent chain* (the canonical backbone determined by GHOSTDAG). Blue blocks are *mergeset blocks*—parallel blocks whose transactions are merged into the ledger when a selected-chain block references them as additional parents.

Recent proposals for Kaspa aimed to address two key issues:

Sequencing Commitment (KIP-15): How to commit each block to the global canonical order of all transactions up to that point, so that Layer-2 (L2) systems can trustlessly follow the exact L1 transaction sequence [1]. KIP-15 proposed adding a chained commitment in each block header that hashes together the previous commitment and the Merkle root of the current block's accepted transactions: $\mathrm{SC}_n = H(\mathrm{SC}_{n-1} \parallel \mathrm{OrderRoot}_n)$. While straightforward, a purely linear commitment implies that to prove a transaction from many blocks ago, one might need to traverse or reveal a long chain of commitments—a verification cost linear in the distance between blocks, which is prohibitive for L2 verification if the distance is large.

Logarithmic Inclusion Proofs (KIP-6): How to enable compact proofs (of size $\mathcal{O}(\log N)$) that a given transaction was included in the ledger by a certain block, without requiring a client to download or trust the entire chain history [2]. KIP-6 recognized that naive proofs of inclusion in Kaspa (or any chain) could be linear in the distance from the block to be verified. The proposal aimed to use an auxiliary data structure or protocol so that one need not traverse every block, reminiscent of the FlyClient approach for light clients.

Each of these alone solves part of the problem, but implementing them separately would mean maintaining two parallel commitment structures: one for ordering and one for efficient proofs. As noted in recent research on Merkle structures [13], a linear hash chain is essentially a degenerate "mountain range" with exactly one peak that grows linearly—maximally unbalanced and far from succinct, forcing proofs to be as long as the list itself.

In this paper, we propose a unified solution that achieves both goals simultaneously by leveraging a data structure called a **Merkle Mountain Range (MMR)** as a historical accumulator of block commitments. MMRs were first popularized by Todd for Bitcoin-like systems [7] as a way to commit to ever-growing data with the ability to produce logarithmic-sized proofs of membership. By replacing the linear sequencing commitment with an MMR commitment, we dramatically improve proof efficiency from $\mathcal{O}(N)$ to $\mathcal{O}(\log N)$.

1.1 Our Contributions

Our approach, which we call the *MMR-based Sequencing Commitment*, integrates with Kaspa's consensus as follows:

- Every block contains a commitment to the set of transactions it accepts (in canonical order) via an `order_root` (replacing the legacy accepted-ID Merkle root in the header).
- Every block also carries an `mmr_root` and `mmr_size`, which accumulate all prior blocks' commitments (leaves) in an append-only Merkle Mountain Range. The MMR acts as a logarithmic-size historical log of the sequence commitments.
- The combination allows any transaction's inclusion position to be proven with two linked proofs: an intra-block Merkle proof to show the transaction is in a block's accepted list, and an MMR proof to show that block's leaf is in the MMR as of some recent block. The latter is of logarithmic size in the number of blocks spanned.

This design effectively provides **everlasting cryptographic receipts** for transactions in Kaspa: a light client or an L2 contract can verify that a transaction was confirmed in block B and at position k in the global order by checking a short proof against a recent block header's MMR root. The proof size and verification cost grow only logarithmically with the age of the transaction, rather than linearly as in a simple chained commitment scheme.

In particular, our contributions and findings include:

1. **Unified Commitment Scheme:** We design a consensus change for Kaspa that introduces two new fields, `order_root` and `mmr_root` (along with a size indicator), into each block header. The `order_root` commits to the block's local transaction order (as per KIP-15), and the `mmr_root` cumulatively includes an entry for every block on the selected chain. This single scheme intertwines KIP-15 and KIP-6 into one mechanism.
2. **Formalization of Canonical Order in a DAG:** We clarify how the canonical transaction order is derived in Kaspa's DAG context, defining $\text{TxList}(B)$ as the exact sequence of accepted transaction IDs determined by consensus, ensuring determinism across all nodes.
3. **MMR Integration and Algorithms:** We describe how MMR leaves are computed and appended using only data available at block validation time (avoiding circular dependencies). We specify the append algorithm and calculation of the MMR root using deterministic bagging of peaks with fixed domain prefixes.

4. **Compact Inclusion Proofs:** We define the structure of MMR inclusion proofs and show that a smart contract or light client can verify a transaction's inclusion in Kaspa's history by: (1) checking the transaction hash against the block's `order_root` via intra-block Merkle proof, and (2) checking the block's leaf against the known `mmr_root` via the MMR proof. Both proofs are logarithmic in size.

5. **Quantitative Analysis:** We demonstrate that verifying a transaction from 100,000 blocks ago requires only ∼17 hash operations in our scheme versus 100,000 sequential hashes in a traditional chain commitment—a 99.98% reduction in verification cost.

1.2 Paper Organization

The rest of this paper is organized as follows. Section 2 provides background on Kaspa's DAG and the need for canonical ordering and efficient proofs, including an introduction to Merkle Mountain Ranges. Section 3 describes the design of the MMR-based sequencing commitment in detail. Section 4 addresses how we handle challenges like block reorganization and pruning. Section 5 discusses the proof construction and verification. Section 6 explores important use cases and benefits. Section 7 analyzes the overhead and performance. Section 8 compares our approach with the separate KIP-15 and KIP-6 schemes. Implementation considerations are given in Section 9. Finally, Section 10 concludes with a summary and implications for future blockchain designs.

2 Background and Motivation

2.1 Blockchain Transaction Ordering and Proofs

In a conventional blockchain (single chain of blocks), establishing the order of transactions and providing proofs of inclusion is straightforward: all blocks form a linear sequence, and within each block a Merkle tree commits to the transactions. A client given a block header (with the Merkle root) and a transaction plus a Merkle branch can verify that transaction was included in that block. To trust that this block is part of the canonical chain, the client must also verify the chain of block hashes (or a proof-of-work difficulty sequence) from a known checkpoint up to that block.

Verifying a transaction from N blocks ago thus typically requires $\mathcal{O}(\log M)$ work for the Merkle proof (where M is the number of transactions in that block) plus $\mathcal{O}(N)$ work to verify the block's inclusion in the chain (unless some accumulator of block headers is used).

Kaspa's blockDAG architecture complicates this in two ways:

1. There isn't a single chain of blocks—blocks can be created concurrently and reference multiple parents. Kaspa's consensus (GHOSTDAG) assigns a partial order to blocks and designates one parent of each block as the *selected parent*.

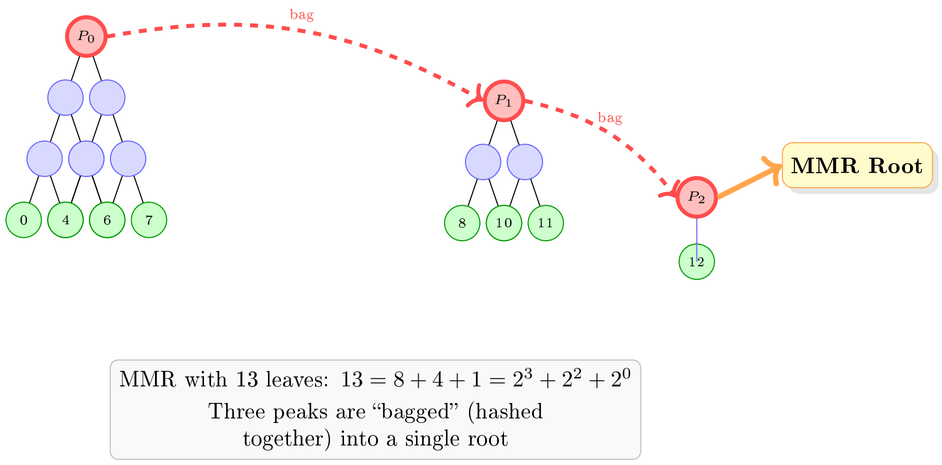
2. Because blocks are not strictly sequential, defining a canonical order of transactions is non-trivial. Kaspa's consensus does produce a deterministic ordering of blocks, but this order must be agreed upon and consistent across all nodes.

The motivation for having an explicit commitment to transaction ordering is largely driven by Layer-2 scalability solutions (like rollups or payment channels) and cross-chain bridges. These systems need to verify L1 events with minimal trust. Without a specialized commitment, verifying an event from an old block might require linear work in the number of blocks since then—impractical on-chain or for light clients.

2.2 Merkle Mountain Ranges (MMR)

Our solution employs Merkle Mountain Ranges (MMRs) as the core data structure to achieve an efficient historical commitment [3]. An MMR is a type of append-only Merkle tree structure especially suited for sequentially growing datasets.

Definition 2.1 (Merkle Mountain Range)**.** *An MMR is defined over an indexed sequence of leaves (data items) numbered from* 0 *onward. It can be thought of as a forest of perfect binary trees ("mountains") whose sizes correspond to the binary decomposition of the leaf count. For N leaves, if $N = \sum_i 2^{b_i}$ in binary, then the MMR consists of trees of sizes 2^{b_i}.*



MMR with 13 leaves: $13 = 8 + 4 + 1 = 2^3 + 2^2 + 2^0$
Three peaks are "bagged" (hashed together) into a single root

Figure 2: Merkle Mountain Range structure with 13 leaves. The structure consists of three "mountains" (perfect binary trees) corresponding to the binary representation $13_{10} = 1101_2$. Peaks P_0, P_1, P_2 are hashed together (bagged) to produce the final MMR root.

Key properties of MMRs that make them attractive for blockchain history accumulators include:

1. **Append-Only and Efficient Updates:** Adding a new leaf to an MMR takes $\mathcal{O}(\log N)$ hashing work. The MMR doesn't require a fixed tree size or re-balancing; it naturally accommodates continuous growth.

2. **Succinct Proofs:** To prove that a given leaf is included in the MMR, one provides a proof consisting of the leaf's sibling nodes up through the tree and the list of peaks. The size of this proof is at most $\mathcal{O}(\log N)$ hashes.

3. **Immutability and Unique Representation:** The structure of the MMR is fully determined by the number of leaves. For a given sequence of N items, the MMR will always produce the same set of peak hashes and the same root.

4. **Inclusion of Historical Data:** Unlike a simple moving hash (like a chain of hashes), the MMR's peaks collectively incorporate all past leaves. Even older blocks far down the "mountain range" are still cryptographically linked into the latest root.

Appending Algorithm: When a new leaf is added to the MMR, it is appended as a new tree of size 1 (a new peak). If the previously last tree (peak) also had size 1, then the two single leaves are merged to form a tree of size 2. This might in turn merge with a next peak of size 2 to form a size 4 tree, and so on. In general, the rule is: upon adding the $(n+1)$-th leaf, look at the binary representation of n. Each trailing 1 in n indicates a peak that will merge with the new leaf (or intermediate result of merges) to form a larger peak. This is equivalent to "carrying" in binary addition.

Example: If $n = 7$ (binary 111, with trailing ones), there were peaks of sizes 1, 2, 4. A new leaf causes the size-1 peak and new leaf to merge (size 2), then that merges with the existing size-2 peak to form size 4, then that merges with the existing size-4 peak to form a size-8 peak. Now for $n + 1 = 8$ (1000_2), there is just one peak of size 8. If $n = 4$ (binary 100, already a perfect tree of size 4 with one peak), adding a leaf results in $n + 1 = 5$ (101_2) which will have two peaks: the old size-4 peak remains and the new leaf stands alone as a size-1 peak.

MMR vs. Linear Chain: A linear hash chain (each block contains the hash of the previous block) is essentially an extreme case of a Merkle structure—a degenerate "mountain range" with exactly one peak that grows linearly. The proof that a given block was in the chain of length N is of length N (one would have to show each link). As noted in recent research [13], the linear chain is maximally unbalanced and far from succinct; it forces proofs to be as long as the list itself. In contrast, an MMR is balanced and yields succinct proofs (logarithmic in N).

2.3 Kaspa DAG, Selected Chain, and Canonical Transaction Order

Kaspa's BlockDAG: Kaspa is a blockDAG where blocks reference multiple parents. This yields high throughput (e.g., 10 blocks per second) but requires a rule to determine a total ordering. Kaspa's consensus (GHOSTDAG) assigns each block a *blue score* and orders blocks by essentially sorting by blue score and hash. For each new block, one of its parents is labeled the *selected parent* (SP)—usually the one with the highest blue score. The set of blocks not on the selected parent chain but referenced by the new block are called the block's *mergeset*.

Accepted Transactions: When a new block arrives, not all transactions in its mergeset are necessarily accepted. If two parallel blocks contain a conflicting transaction (double spend), only the transactions from the block that is "preferred" by GHOSTDAG (the blue set) will be accepted. However, all non-conflicting transactions from all blocks will be included in the ledger's UTXO state.

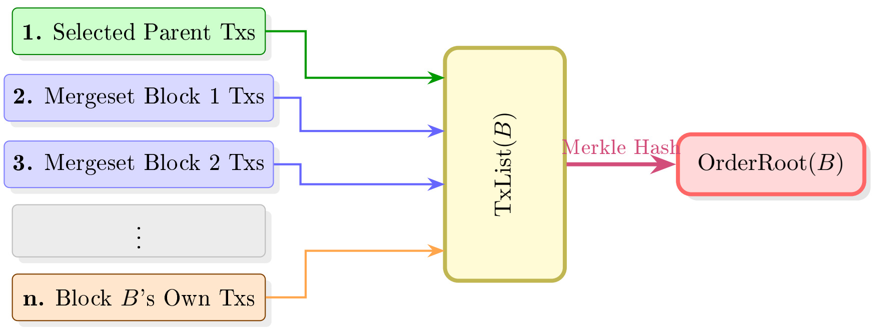
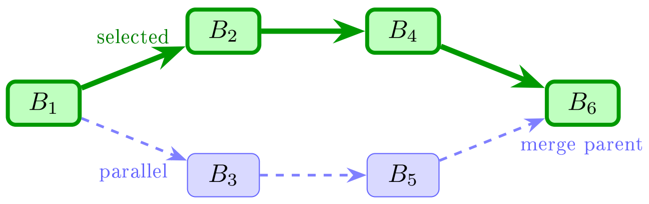

1. Selected Parent Txs
2. Mergeset Block 1 Txs
3. Mergeset Block 2 Txs
⋮
n. Block B's Own Txs
TxList(B)
Merkle Hash
OrderRoot(B)

Figure 3: Construction of the canonical transaction list (TxList) and OrderRoot for block B. Transactions are ordered deterministically: selected parent first, then mergeset blocks in consensus order, finally block B's own transactions.

Kaspa's node implementation computes, for each block B, the set of *accepted transaction IDs*—essentially the union of the transactions in the block's selected parent and those in the block's remaining mergeset (excluding conflicts), in a deterministic order. By walking through the DAG following the selected chain and at each block outputting its accepted transaction list, one obtains a single global sequence of transactions. This global sequence is what an L2 system would interpret as the history of L1 transactions.

Detailed Example: Consider a simple DAG scenario for illustration. Suppose block B_1 is the genesis, and two blocks B_2 and B_3 are mined concurrently on B_1. Then a block B_4 comes selecting B_2 as its parent, and another B_5 selecting B_3, and finally B_6 selecting B_4 and also referencing B_5:



Here the selected parent chain is: $B_1 \to B_2 \to B_4 \to B_6$ (green blocks). The canonical transaction sequence up to B_6 would then be:

1. TxList(B_1): all transactions from genesis
2. TxList(B_2): transactions from B_2
3. Transactions from B_3 not already included (merged when B_4 or B_6 processes its mergeset)
4. TxList(B_4): transactions from B_4
5. Transactions from B_5 not already included (merged when B_6 is processed)
6. TxList(B_6): transactions from B_6

This results in one total order of all accepted transactions up to block B_6. If two transactions conflict (double-spend), Kaspa's consensus ensures only one will be accepted—generally the one appearing in the earlier position in the canonical order.

KIP-15's originally suggested mechanism was to maintain a running hash chain: $SC_n = H(SC_{n-1} \parallel \text{OrderRoot}(B_n))$. While this achieves commitment to the sequence, it inherits a major limitation: verifying that a given old block's transactions are in the chain up to some later point requires iterating through each block's hash link in between (linear complexity). Our approach replaces this linear chain of commitments with the MMR accumulator, which breaks the sequential dependency and allows skip-ahead verification.

3 MMR-Based Sequencing Commitment Design

In this section, we present the detailed design of the MMR-based sequencing commitment. We describe the modifications to the Kaspa block header, define how the MMR is computed and updated with each new block, and specify the consensus rules for validation.

3.1 Block Header Modifications

We extend the Kaspa block header with two new fields and modify one existing field:

At 10 blocks/sec, the net increase is ∼34.6 MB per day, which is negligible overhead for network transmission (roughly 0.08% of typical block data throughput).

3.2 Canonical Order and OrderRoot Computation

Definition 3.1 (Canonical Transaction List). *For each block B, we define an ordered list of transaction IDs, denoted* TxList(B)*, which is the sequence of all accepted transactions that are considered to "happen at" block B in the global order. This list is constructed from the block's selected parent and mergeset in the consensus processing.*

Suppose block B has selected parent P_{sel} and a set of merge parents $\{M_1, M_2, \ldots, M_k\}$. The UTXO validation algorithm in Kaspa will process:

1. The coinbase transaction of P_{sel} (if not already included earlier)

2. The remaining transactions of P_{sel}, in their original order

3. The transactions from each merge block M_i in a deterministic order

Table 1: Block Header Field Modifications

Field	Size	Description
`order_root`	32 bytes	Merkle root of accepted transaction IDs for this block, representing the set of transactions (in canonical order) confirmed by this block's addition to the ledger
`mmr_root`	32 bytes	MMR root hash after inserting this block's leaf, serving as a cryptographic commitment to the entire history of transaction order
`mmr_size`	8 bytes	Total number of leaves in the MMR after adding this block
Removed field:		
`accepted_id_merkle_root`	32 bytes	(Replaced by `order_root`)
Net Change	**+40 bytes**	($2 \times 32 + 8 - 32 = 40$ bytes per header)

4. Transactions in B itself

We define:

$$\text{OrderRoot}(B) := \text{MerkleRoot}(\text{TxList}(B)) \tag{1}$$

Determinism and Security: Because all honest nodes will derive the same TxList for B, the OrderRoot is an objective, deterministic value. If a malicious miner tried to put a different OrderRoot in the header that doesn't match the actual transactions, consensus validation will catch it.

3.3 MMR Leaf and Internal Hashing Scheme

For each block B that lies on the selected parent chain, we create an MMR leaf node:

$$L(B) = \text{H}_{\text{leaf}}(h_{P_{\text{sel}}(B)} \parallel \text{OrderRoot}(B)) \tag{2}$$

where $h_{P_{\text{sel}}(B)}$ is the hash of B's selected parent block header, and H_{leaf} is a hash function instance domain-separated for leaf hashing.

Rationale for leaf contents:

- We include the selected parent's hash to anchor this leaf in the chain of blocks. This ensures that if a block is reordered or placed under a different parent, its leaf hash would change.
- We include the OrderRoot to commit to the transaction content.
- We do *not* include the block's own hash in the leaf to avoid a circular dependency: the block's hash itself is computed from the header fields, which include the MMR root.

MMR Structure Maintenance: The MMR is maintained as follows. Let $S(n)$ denote the state of the MMR after processing n leaves. When a new leaf arrives:

1. We initially consider the new leaf as a new peak (a tree of size 1)
2. As long as we have two peaks of equal size, we merge them (like adding binary numbers)
3. The number of merges equals the number of trailing 1 bits in the old size

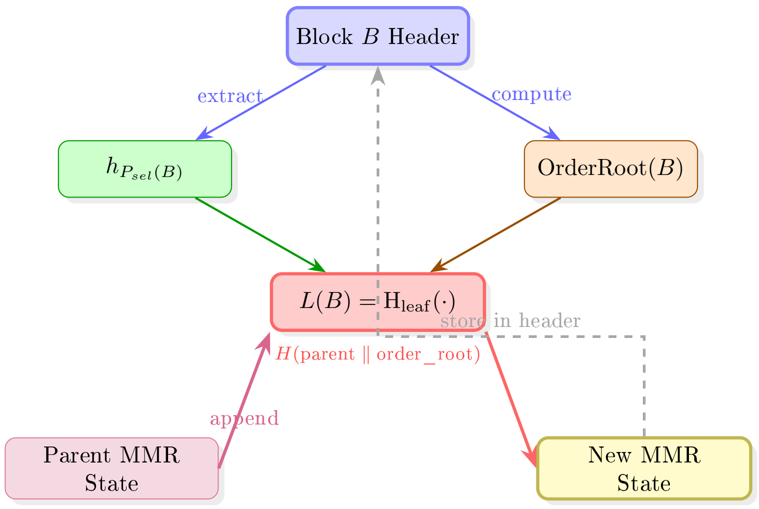


Figure 4: MMR leaf computation and state update flow. The leaf is computed from parent hash and OrderRoot, then appended to the parent's MMR state to produce the new MMR root stored in the block header.

Internal Node Hashing: When two peaks are merged, we use a domain-separated hash:

$$\text{merged} = \mathrm{H}_{\text{node}}(\text{left\_peak} \parallel \text{right\_peak}) \tag{3}$$

Peak Bagging to Root: After processing the new leaf and merging as needed, we compute the final MMR root by hashing all peaks together ("bagging the peaks"):

$$\text{bag}_0 = p_0 \tag{4}$$
$$\text{bag}_i = \mathrm{H}_{\text{bag}}(\text{bag}_{i-1} \parallel p_i) \quad \text{for } i = 1, \ldots, k-1 \tag{5}$$
$$\text{mmr\_root} = \mathrm{H}_{\text{root}}(\text{bag}_{k-1} \parallel \text{encode}_{64}(\text{mmr\_size})) \tag{6}$$

Including the size in the final hash ensures that different sequences of leaves cannot produce the same root. The order of peaks and the hashing procedure must be canonical and agreed upon in consensus. In our design, peaks are ordered from smallest-index (leftmost, corresponding to largest tree) to largest-index (rightmost, possibly a small tree) and hashed in that sequence.

The TrailingOnes function returns the number of consecutive 1 bits in the binary representation starting from the least significant bit. For example, TrailingOnes(7) = 3 (binary 111) and TrailingOnes(4) = 0 (binary 100). This achieves $\mathcal{O}(1)$ amortized time per append with a small constant factor for the merging.

3.4 Consensus Validation Rules

When processing block B, nodes perform the following validations:

The MMR verification adds a small constant amount of work per block: computing one Merkle root (for transactions) and a handful of hashes for the MMR update. On average, this is less than 2 hashes per block, and even in worst-case bursts (when block height is a power of 2 minus 1), the hashing overhead is negligible relative to mining and other validation.

4 Handling Reorganizations and Forks

One of the challenges in a blockchain (especially a DAG-based one) is handling chain reorganization (reorg) events, where the selected parent chain may change. Kaspa's GHOSTDAG can cause the selected chain to change at the virtual block if a fork is resolved in favor of a different chain of blue blocks.

Algorithm 1 MMR Append Operation

```
Require: Current MMR state (peaks list, size), new leaf hash
Ensure: Updated MMR state, new root
 1: function APPEND(state, new_leaf)
 2:     new_size ← state.size +1
 3:     current_hash ← new_leaf
 4:     new_peaks ← copy(state.peaks)
 5:     count ← TRAILINGONES(state.size)  ▷ Number of trailing 1-bits in binary representation
 6:     for $i$ ← 1 to count do
 7:         left_peak ← new_peaks.pop()
 8:         current_hash ← $H_{node}$(left_peak || current_hash)
 9:     end for
10:     new_peaks.append(current_hash)
11:     bag ← new_peaks[0]
12:     for $i$ ← 1 to new_peaks.length −1 do
13:         bag ← $H_{bag}$(bag || new_peaks[$i$])
14:     end for
15:     mmr_root ← $H_{root}$(bag || $encode_{64}$(new_size))
16:     return (MmrState{peaks: new_peaks, size: new_size}, mmr_root)
17: end function
```

4.1 Reorg Rollback and Recompute

In the event of a reorg:

1. Let C be the last common ancestor block between the old and new selected parent chain (the fork point)

2. The old chain had leaves for blocks after C that are no longer on the selected chain

3. The new chain has a sequence of blocks from C to the new tip that were not previously on the main chain

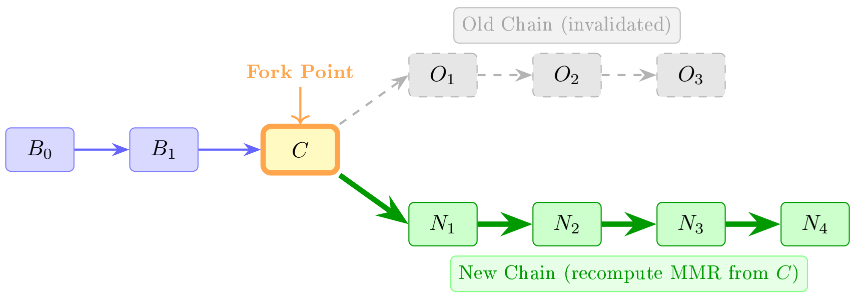


Figure 5: Chain reorganization handling. MMR state is retrieved from common ancestor C, and new chain blocks have their MMR states recomputed forward.

To handle this:

- We maintain the MMR state per block (peaks and size) in our store

- During reorg, we find C (the common ancestor) and retrieve C's MMR state

- We iterate forward from C along the new selected chain, recomputing the MMR state for each block

Reorg handling is $\mathcal{O}(k)$ where k is the number of blocks switched, typically small.

Algorithm 2 Block Validation with MMR Commitments

```
Require: Block $B$ with header fields, Parent block $P$
Ensure: Block validity
 1: Step 1: Order-Root Verification
 2: expected_order_root $\leftarrow$ MerkleRoot(accepted_tx_ids of $B$)
 3: if expected_order_root $\neq B$.order_root then
 4:     return BadOrderRoot
 5: end if
 6: Step 2: MMR Leaf Computation
 7: $\text{leaf}_B \leftarrow \text{H}_{\text{leaf}}(P.\text{hash} \parallel B.\text{order\_root})$
 8: Step 3: MMR Append Computation
 9: old_size $\leftarrow P$.mmr_size
10: new_size $\leftarrow$ old_size $+1$
11: peaks $\leftarrow P$.mmr_peaks
12: current $\leftarrow \text{leaf}_B$
13: while old_size has trailing 1 bit do
14:     left_peak $\leftarrow$ peaks.pop()
15:     current $\leftarrow \text{H}_{\text{node}}(\text{left\_peak} \parallel \text{current})$
16:     old_size $\leftarrow$ old_size $\gg 1$
17: end while
18: peaks.push(current)
19: Step 4: Compute Expected MMR Root
20: bag $\leftarrow$ peaks[0]
21: for $i \leftarrow 1$ to peaks.length $-1$ do
22:     bag $\leftarrow \text{H}_{\text{bag}}(\text{bag} \parallel \text{peaks}[i])$
23: end for
24: expected_mmr_root $\leftarrow \text{H}_{\text{root}}(\text{bag} \parallel \text{encode}_{64}(\text{new\_size}))$
25: Step 5: Verification
26: if expected_mmr_root $\neq B$.mmr_root then
27:     return BadMmrRoot
28: end if
29: if new_size $\neq B$.mmr_size then
30:     return BadMmrSize
31: end if
32: return Valid
```

4.2 MMR Node Storage (Enabling Proof Generation)

While the peaks alone are enough for consensus validation (we just need to recompute the new root), generating an actual inclusion proof for an arbitrary past block or transaction requires access to the intermediate nodes (siblings along the path). A full node that wants to support light clients or bridges with proofs should maintain an **MMR Node Store** mapping each node position to its hash.

Indexing Scheme: The positions can be indexed by a single number if we linearize the MMR tree in a certain traversal. A simple scheme is:

- Label the leaves with indices starting from 0
- Label internal nodes in the order they are created during the append process
- The position of a node can be derived from the leaf index and the node's height

We store each node hash keyed by its position. To produce a proof for leaf i, we:

1. Compute the positions of each sibling on the path from that leaf to the peak
2. Retrieve those sibling hashes from the store
3. Also retrieve the peak hashes

All of this is $\mathcal{O}(\log n)$ operations.

Storage Cost: Each leaf or internal node is one hash (32 bytes). An MMR with n leaves has $< 2n$ nodes in total (similar to a full binary tree). The node store size for n leaves is roughly $2n \times 32$ bytes:

- After 1,000,000 leaves: $2 \times 10^6 \times 32 = 64$ MB
- After 300,000,000 leaves (∼1 year at 10 BPS): ∼19.2 GB

This is within reason for a well-resourced node serving many light clients.

Recommended Storage Practice: For a full node at 10 BPS:

1. Store all block headers permanently (few tens of GB/year)—required for consensus
2. Maintain an MMR node store persistently (∼20 GB/year) if supporting historical proofs
3. Maintain a rolling cache of peaks for recent blocks (last day or week) for fast reorg handling. For example, 86,400 blocks for a day at 10 BPS with ∼20 peaks each ≈ 55 MB

4.3 Pruning and Checkpointing

Kaspa nodes may prune old blocks, but they typically keep all headers (including the MMR fields). Thus, a pruned node can still verify a proof of inclusion for a pruned block: it has that block's header and the tip header.

A headers-only node can verify inclusion of a transaction from any block, given a correct proof. This is by design: enabling light clients to be almost as secure as full nodes for checking inclusion, without storing everything. The MMR commitment turns the whole history into a *self-authenticating log*: the node can verify any historical data from an untrusted source against the known `mmr_root`.

4.4 Rolling MMR for Long-Term Scalability

While an append-only MMR provides excellent properties for historical verification, the unbounded growth of the MMR poses long-term storage challenges. At 10 BPS, the MMR node store grows at approximately 20 GB per year. Over decades, this becomes significant even for well-resourced nodes. We propose a **Rolling MMR** mechanism to address this:

4.4.1 Checkpoint-Based MMR Rolling

Rather than maintaining the complete MMR from genesis indefinitely, nodes can periodically "roll forward" the MMR base:

1. **Checkpoint Selection:** At regular intervals (e.g., every pruning epoch, approximately 24 hours), designate a checkpoint block B_c on the selected parent chain
2. **MMR State Snapshot:** Store the complete MMR peaks at B_c as a *frozen checkpoint*
3. **Prefix Pruning:** Old MMR nodes (those corresponding to leaves before B_c) can be pruned, keeping only:
    - The frozen checkpoint state (peaks and size at B_c)
    - Recent MMR nodes (from B_c onward)

4.4.2 Proof Generation with Rolling MMR

For transactions in blocks *after* the checkpoint B_c, proof generation works normally using the local MMR store.

For transactions in blocks *before* B_c:

- The proving node must either (a) have retained the full historical MMR, or (b) request the historical proof from an archive node
- The proof can be composed: a proof from the old transaction to the checkpoint's MMR root, plus a proof from the checkpoint to the current tip
- Alternatively, users who need long-term proofs should generate and store them *before* the relevant MMR nodes are pruned

4.4.3 Storage Trade-offs

Strategy	**Storage (per year)**	**Proof Availability**
Full MMR (no rolling)	∼20 GB/year cumulative	All historical proofs
Daily checkpoints	∼500 MB/year	Recent + checkpoint-based
Weekly checkpoints	∼100 MB/year	Recent + archived

The rolling approach is optional: archive nodes can maintain the complete MMR history, while regular nodes use rolling to bound storage. This mirrors how Kaspa already handles block body pruning—the MMR commitment in headers enables verification even when underlying data is pruned, as long as proofs were generated before pruning.

4.4.4 Integration with Kaspa's Pruning Mechanism

Kaspa already prunes block bodies after a configurable period (currently 3 days by default). The rolling MMR can be synchronized with this mechanism:

- **Pre-pruning proof generation:** Nodes can automatically generate and cache inclusion proofs for their own transactions before pruning
- **Checkpoint alignment:** MMR checkpoints can align with Kaspa's posterity headers (every ∼86,400 blocks)
- **Archive incentives:** Archive nodes maintaining full MMR history can charge for historical proof generation, creating a sustainable ecosystem

This design ensures that the MMR-based system remains practical at scale while preserving the cryptographic guarantees for any historical data that has been properly archived or proven.

5 Inclusion Proofs and Verification

A major goal of this design is to enable efficient transaction inclusion proofs that a light client or smart contract can verify.

5.1 Proof Composition

To prove that a transaction tx was included in the Kaspa ledger, we need to provide evidence for two things:

1. **Transaction-in-Block:** tx is included in block B's accepted transaction list at a certain index
2. **Block-in-Ledger:** Block B's commitment leaf is included in the MMR up to some known recent block T

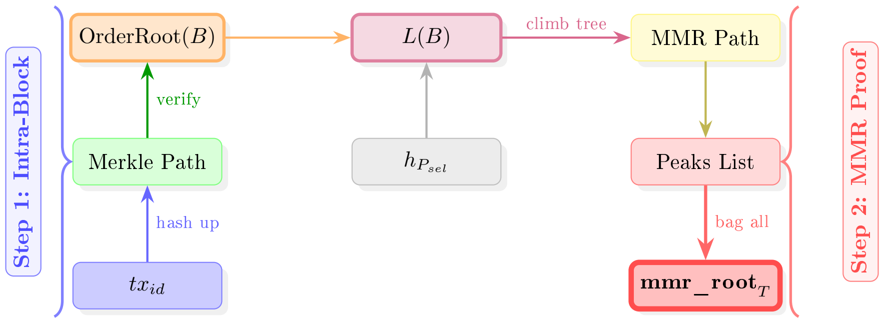


Figure 6: Two-step proof verification process. Step 1 verifies the transaction is in block B via Merkle proof. Step 2 verifies block B is in the MMR via MMR proof.

We formalize a proof as a tuple $(txId, B, \text{IntraProof}, \text{MMRProof}, T)$ where:

- $txId$ is the transaction identifier (hash) we are proving
- B is the block hash in which tx was included
- IntraProof contains the Merkle path nodes needed to compute B's OrderRoot from $txId$
- MMRProof contains the MMR leaf index, sibling hashes, and peaks list
- T is the block hash whose MMR root we are anchoring to

Intra-Block Merkle Proof Format: This is the classic Merkle branch: given the index of the transaction in the block and the Merkle root (`order_root`) from the block header, provide the sibling hashes at each level:

```
IntraProof = {
    tx_index: u32,              # Position in TxList
    siblings: [(Hash, bool)],   # (sibling_hash, is_left)
}
```

Listing 1: Intra-Block Proof Structure

MMR Inclusion Proof Format:

```
MMRProof = {
    leaf_index: u64,            # 0-based leaf index
    mmr_size: u64,              # Size at proof time
    siblings: [Hash],           # Path from leaf to peak
    peaks: [Hash],              # All peak hashes
    parent_hash: Hash,          # B's selected parent hash
    order_root: Hash,           # B's order_root
}
```

Listing 2: MMR Proof Structure

5.2 Proof Size and Efficiency

Theorem 5.1 (Proof Size Bound)**.** *For a transaction in block B with m transactions, and MMR with N leaves, the total proof size is:*

$$\mathit{ProofSize} = \mathcal{O}(\log m + \log N) \times 32 \text{ bytes} \tag{7}$$

Table 2: Proof Size Estimates for Various Chain Lengths at 10 BPS

Chain Length	Time	MMR Depth	Est. Proof Size	Verification Hashes
10^6 blocks	∼1 day	20	∼800 bytes	∼25
10^7 blocks	∼12 days	24	∼900 bytes	∼30
10^8 blocks	∼4 months	27	∼1.0 KB	∼35
3×10^8 blocks	∼1 year	28	∼1.1 KB	∼40

This is a huge improvement over a linear chain proof, which would require every block header hash in between (easily tens of kilobytes or more for deep history).

Table 3: Proof Size (Hash Operations) vs Age of Transaction

Blocks Ago (d)	KIP-15 Hash Ops	MMR Hash Ops	Improvement
100	∼100	∼7–8	93% fewer
1,000	∼1,000	∼10–11	99% fewer
10,000	∼10,000	∼13–14	99.9% fewer
100,000	∼100,000	∼17–18	99.98% fewer
1,000,000	∼1,000,000	∼20–21	99.998% fewer

Even at moderate distances like 10,000 blocks (∼1,000 seconds at 10 BPS), the difference is stark: 10,000 vs ∼14 operations. This demonstrates that our approach is not just asymptotically better, but very practically so, enabling use cases that were completely out of reach before (e.g., verifying a 1-year old transaction on-chain is impossible linearly but easy with MMR).

On-chain Verification Gas Costs: On-chain verification (e.g., in an Ethereum contract) of ∼30 hashes is feasible. Each hash costs approximately ∼1920 gas (for Keccak-256: ~ 8 gas/byte $\times$ 64 bytes). Therefore:

- MMR proof (30 hashes): ∼57,600 gas—very manageable (typical ETH transfer costs ∼21,000 gas)
- Linear 100k hash proof: ∼192 million gas—exceeds block gas limit (∼30M) by 6×

Batch Verification Advantage: One great advantage of MMRs is that proofs for multiple leaves can share parts. If an L2 contract needs to verify k transactions from possibly different blocks:

Algorithm 3 Complete Proof Verification

```
Require: Transaction ID $txId$, IntraProof, MMRProof, trusted mmr_root$_T$
Ensure: Proof validity (true/false)
 1: Step 1: Verify Transaction in Block
 2: hash ← $txId$
 3: for (sibling, is_left) in IntraProof.siblings do
 4:     if is_left then
 5:         hash ← $H$(sibling ‖ hash)
 6:     else
 7:         hash ← $H$(hash ‖ sibling)
 8:     end if
 9: end for
10: if hash ≠ MMRProof.order_root then
11:     return FALSE
12: end if
13: Step 2: Compute MMR Leaf
14: leaf ← $H_{leaf}$(MMRProof.parent_hash ‖ MMRProof.order_root)
15: Step 3: Verify MMR Path to Peak
16: current ← leaf
17: pos ← MMRProof.leaf_index
18: for sibling in MMRProof.siblings do
19:     if pos is even then                                   ▷ Current is left child
20:         current ← $H_{node}$(current ‖ sibling)
21:     else                                                  ▷ Current is right child
22:         current ← $H_{node}$(sibling ‖ current)
23:     end if
24:     pos ← pos / 2
25: end for
26: peak ← current
27: Step 4: Verify Against MMR Root
28: bag ← None
29: for p in MMRProof.peaks do
30:     if bag is None then
31:         bag ← p
32:     else
33:         bag ← $H_{bag}$(bag ‖ p)
34:     end if
35: end for
36: expected_root ← $H_{root}$(bag ‖ encode$_{64}$(MMRProof.mmr_size))
37: return expected_root = mmr_root$_T$
```

- If from the same block: they share the entire MMR path
- If from different blocks: they share the peaks list
- Even 100 separate proofs might total only ∼3M gas, which is acceptable

This is crucial for verifying multiple withdrawals in one go on a bridge.

6 Applications and Benefits

With the MMR-based sequencing commitment in place, several important use cases become possible:

6.1 Trustless Bridges and L2 Rollups

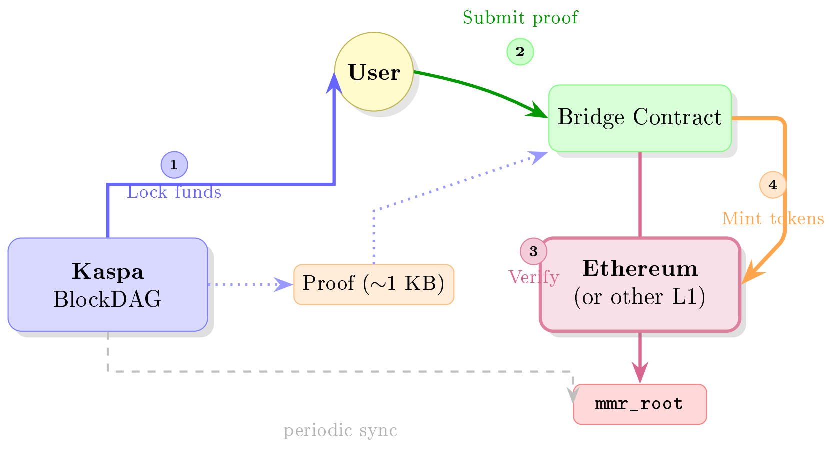


Figure 7: Trustless bridge architecture using MMR proofs. The bridge contract stores only the latest `mmr_root` and verifies inclusion proofs on-chain with ∼30 hash operations.

One primary driver for this feature is to facilitate L2 rollups and cross-chain bridges. With our scheme, an Ethereum contract can store just a single 32-byte `mmr_root` and for each verification, accept a proof and only do $\mathcal{O}(\log N)$ hashing.

Furthermore, MMR proofs support batch verification optimization. Multiple transactions can share parts of the proof path, and the peaks list and top-level bagging is common across all proofs. This drastically reduces the risk of bridges—which, as noted by Chainalysis [14], have been a major source of hacks when relying on trusted parties.

Quantitative Gas Comparison for Cross-Chain Verification:

Table 4: On-Chain Verification Cost Comparison (Ethereum)

Proof Depth	KIP-15 Gas	MMR Gas	Feasibility
100 blocks	192,000	∼15,000	Both OK
1,000 blocks	1,920,000	∼21,000	KIP-15 borderline
10,000 blocks	19,200,000	∼27,000	KIP-15 fails
100,000 blocks	192,000,000	∼35,000	KIP-15 impossible
1,000,000 blocks	1,920,000,000	∼42,000	KIP-15 impossible

At 100,000 blocks depth (∼2.8 hours at 10 BPS), KIP-15 exceeds the block gas limit by 6×, while MMR verification costs less than 0.1% of the block gas limit.

6.2 Light Clients and Fast Synchronization

For Kaspa's own ecosystem, having cryptographic receipts means:

- **Simple Payment Verification (SPV):** A client only needs to track the chain of `mmr_root` in headers and for any transaction of interest, get a proof from a peer and verify it
- **Checkpoint Jumping:** A new node can perform randomized checks against the tip's MMR root
- **Parallelized Download:** Blocks can be downloaded out-of-order and verified in parallel
- **On-demand Historical Retrieval:** Pruned nodes can fetch old data with verifiable proofs

Skip Verification Strategies: With an MMR, a node can employ several advanced sync techniques:

1. **Checkpoint Jump:** The node accepts a recent `mmr_root` from a trusted checkpoint or the network. It then downloads blocks in bulk (from any source, even untrusted peers) and verifies each block's `order_root` against its header and then verifies that block's inclusion in the MMR. If any block's data was tampered with, its `order_root` would not match, or its leaf proof would fail. The node can download blocks in parallel and check them in any order—the final `mmr_root` ties them all together.
2. **Sampling for Security:** If a node doesn't fully trust a given source, it can do random sampling. For example, obtain all block headers from the network (relatively light, just a few GB), thus obtaining the final `mmr_root`. Then download actual transactions for say 1 out of each 100 blocks (randomly chosen), verify those blocks' `order_roots` and MMR proofs. If all samples check out, it has high confidence the history is genuine. This drastically reduces work—checking 1% of the blocks rather than 100%.
3. **Parallel Block Verification:** Even for full verification, "integrity verification" can be decoupled from UTXO validation. Multiple threads can verify each block's inclusion independently, sharing the final `mmr_root` as a common reference.

6.3 Data Availability and Disaster Recovery

A subtle benefit of having an on-chain history commitment is in data availability and recovery. If historical blocks become unavailable (many nodes pruned and only few archives remain), one might worry about losing that data. With our scheme, as long as the `mmr_root` is known, anyone who eventually finds or reconstructs the block can prove its correctness.

Recovery Process:

1. Retrieve recent block headers from peers (which give the recent `mmr_root`)
2. Start from the last trusted checkpoint (e.g., stored MMR state at pruning point)
3. For each needed block, fetch from any source (could be a torrent, centralized archive, etc.)
4. Verify block's `order_root` and then verify its inclusion via MMR proof to the known latest root
5. If all checks pass, safely insert the block into local history

This process ensures that no malicious or corrupted data sneaks in during recovery. The combination of the on-chain commitment and the proofs turns the blockchain history into a *self-verifiable log*—any single block's wrong data breaks the root instantly, strengthening security for archival data.

6.4 Auditing and Data Availability

Because the MMR ensures that any omitted or altered transaction would change the root, it provides a solid foundation for audit tools. An auditor could store just the MMR root of each day's tip and later verify any claimed transaction set against it.

7 Overhead Analysis

7.1 Bandwidth Overhead

Table 5: Bandwidth Overhead at 10 BPS

Metric	**Value**
Additional header bytes	40 bytes/block
Bandwidth at 10 BPS	400 bytes/sec
Daily overhead	∼34.6 MB
Yearly overhead	∼12.6 GB
Relative to block data (∼50 KB/block)	$< 0.1\%$

7.2 Storage Overhead

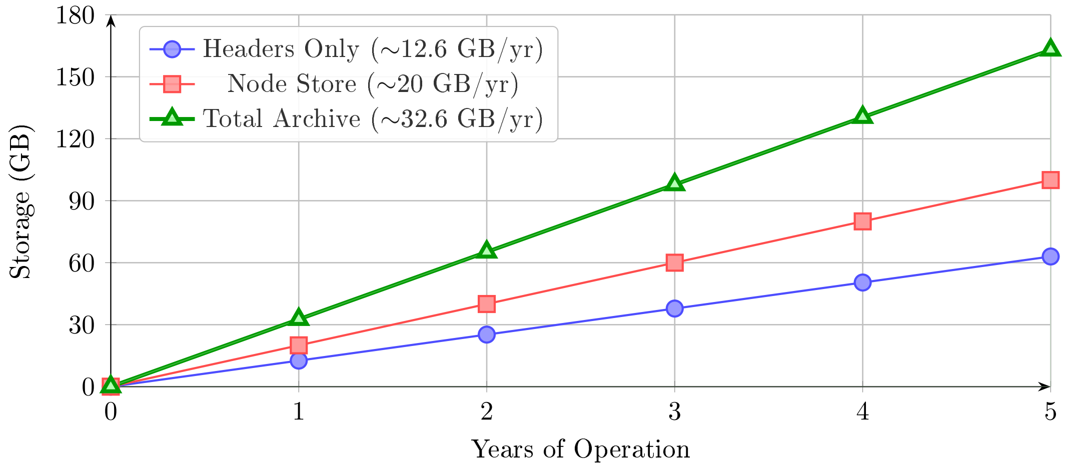


Figure 8: Storage growth over time at 10 BPS. Headers grow at ∼12.6 GB/year; optional node store for proof generation adds ∼20 GB/year. Even after 5 years, total storage remains under 165 GB—manageable for modern hardware.

7.3 Computation Overhead

Per-block hashing operations:

- Merkle root computation: $\mathcal{O}(\log m)$ hashes (same as before)
- MMR update: Average < 2 hashes, worst case $\sim \log N$ hashes
- Total: Negligible compared to PoW and signature verification

The trade-off is extremely favorable: for a modest storage and negligible performance cost, Kaspa gains a built-in universal proof system for its transaction history.

8 Comparison with Alternative Approaches

In this section, we provide a detailed comparison between our MMR-based unified approach and the two Kaspa Improvement Proposals that inspired it: KIP-6 (Proof of Chain Membership) and KIP-15 (Canonical Transaction Ordering and Sequencing Commitment). Understanding the design philosophy, strengths, and limitations of each approach illuminates why our unified solution represents an optimal path forward.

8.1 KIP-15: Linear Sequencing Commitment

8.1.1 Design Philosophy and Mechanism

KIP-15, authored by Mike Zak and Ro Ma [1], addresses a fundamental need for Layer-2 networks: the ability to trustlessly verify the *order* of transaction acceptance on L1. The proposal recognizes that L2 networks rely on Kaspa for both consensus and data availability, where "the ordering of transaction acceptance in L1 has to be the ordering of transaction execution on L2."

The core mechanism introduces a new header field called `SequencingCommitment`, calculated as:

$$\mathrm{SC}_B = H(\mathrm{SC}_{P_{sel}(B)} \parallel \mathrm{AcceptedIDMerkleRoot}_B) \tag{8}$$

where $P_{sel}(B)$ is the selected parent of block B.

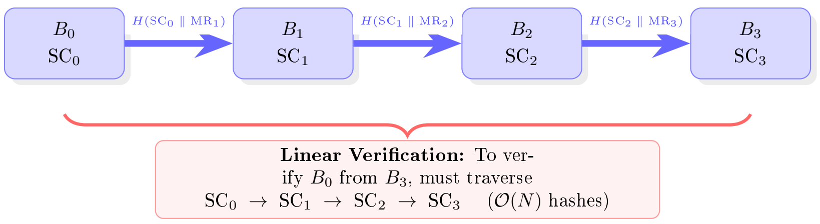

B_0 SC_0
$H(\mathrm{SC}_0 \parallel \mathrm{MR}_1)$
B_1 SC_1
$H(\mathrm{SC}_1 \parallel \mathrm{MR}_2)$
B_2 SC_2
$H(\mathrm{SC}_2 \parallel \mathrm{MR}_3)$
B_3 SC_3
Linear Verification: To verify B_0 from B_3, must traverse $\mathrm{SC}_0 \rightarrow \mathrm{SC}_1 \rightarrow \mathrm{SC}_2 \rightarrow \mathrm{SC}_3$ ($\mathcal{O}(N)$ hashes)

Figure 9: KIP-15 Linear Sequencing Commitment. Each block's SC depends on its parent's SC, creating a sequential chain. Verification requires traversing all intermediate commitments.

Additionally, KIP-15 proposes a critical change: the `AcceptedIDMerkleRoot` should preserve *canonical order* rather than sorting transactions by hash. The original hash-sorting was an optimization for exclusion proofs, but as KIP-15 notes: "there is currently no application on top of Kaspa using the above feature."

8.1.2 Strengths

1. **Simplicity:** One additional hash per block; minimal implementation complexity
2. **Strong Ordering Guarantee:** The chain of commitments cryptographically binds the exact sequence of all accepted transactions
3. **ATAN Support:** Enables Accepted Transactions Archival Nodes to bootstrap from untrusted sources with verification
4. **Low Header Overhead:** Replaces existing field rather than adding new ones (net change: 0 bytes)

8.1.3 Limitations

1. **Linear Verification Complexity:** To verify that a transaction in block N is part of the history up to block M, one must compute or provide all $M-N$ intermediate commitments. As stated in the KIP: "downloading and validating a cryptographic proof testifying to the above" requires processing the entire selected parent chain.

2. **No Skip-Ahead Capability:** The sequential dependency prevents any form of random access or parallel verification.

3. **Impractical for Bridges:** On-chain verification (e.g., in an Ethereum smart contract) of a proof spanning 100,000 blocks would require ∼100,000 hash operations—exceeding any reasonable gas limit by orders of magnitude.

4. **No Independence from KIP-6:** KIP-15 alone does not provide efficient proofs of chain membership; the document's ATAN design assumes full header availability.

8.2 KIP-6: Proof of Chain Membership (PoChM)

8.2.1 Design Philosophy and Mechanism

KIP-6, authored by Shai Wyborski [2], addresses a complementary problem: providing cryptographically verifiable proofs that a transaction was included in the blockchain, especially after pruning has occurred. The proposal recognizes that "the pruning mechanism makes it impossible to prove that a transaction was included in the ledger after it has been pruned" and that relying on centralized archival nodes "is not a sustainable solution."

KIP-6 introduces the concept of **Posterity Headers**—selected chain block headers sampled at regular intervals (currently every 24 hours, coinciding with pruning blocks) and stored indefinitely. The core mechanism adds a **PoChM Merkle Root (PMR)** to each header:

Definition 8.1 (PMR Construction (KIP-6))**.** *Let k be the least integer such that $parent(B, 2^k) \in Past(next\_posterity(B))$. Then PMR is the root of the Merkle tree containing the headers $parent(B, 2^i)$ for $i = 0, \ldots, k-1$.*

This creates a skip-list-like structure where each block commits to ancestors at exponentially increasing distances $(2^0, 2^1, 2^2, \ldots)$.

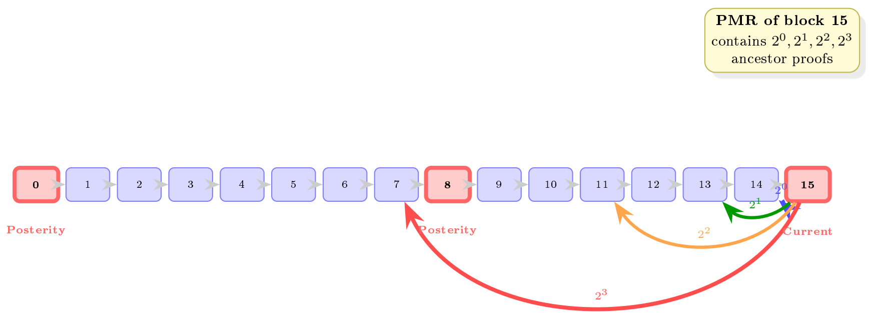


Figure 10: KIP-6 PoChM Structure with exponential skip pointers (2^i). Each block's PMR contains Merkle proofs of ancestors at distances $2^0, 2^1, 2^2, \ldots$ The proof path uses binary decomposition of the distance.

8.2.2 Proof Construction Algorithm

The PoChM generation algorithm (from KIP-6) uses binary decomposition:

1. Let $C = \text{posterity}(B)$ and $d = \text{posterity\_depth}(B)$

2. While $d > 0$:

   - Let $i = \lfloor \log_2(d) \rfloor$
   - Append $\text{PMR}(C, i)$ to proof

- $d \leftarrow d - 2^i$; $C \leftarrow \text{parent}(C, 2^i)$

This yields proof complexity of $\mathcal{O}(\log N \cdot \log\log N)$ where N is the number of chain blocks between posterity headers.

8.2.3 Strengths

1. **Logarithmic Proofs:** Proof size is $\mathcal{O}(\log N \cdot \log\log N)$, approximately 9 KB worst case
2. **Minimal Header Overhead:** Only 32 bytes (one hash) added per header
3. **Efficient Validation:** 33 hashes per block for chain blocks
4. **Pruning Compatible:** Proofs can be generated before pruning and verified indefinitely
5. **Elegant Binary Decomposition:** Mathematically principled approach leveraging number-theoretic structure

8.2.4 Limitations

1. **Does Not Commit Transaction Order:** KIP-6 provides proofs that a block was in the selected chain, but not the canonical ordering of transactions. As stated: "a PoChM for B is also a PoChM for any chain block C ∈ Future(B) ∩ Past(posterity(B))"—the proof is about chain membership, not transaction sequence.
2. **Requires Separate Structure for Ordering:** To prove both chain membership *and* transaction order, KIP-6 must be combined with KIP-15 or similar, resulting in two parallel commitment structures.
3. **Posterity Header Dependency:** The scheme depends on posterity headers (currently 24-hour intervals). This introduces latency: a PoChM cannot be generated until the next posterity header exists.
4. **Complex Proof Structure:** The $\log\log N$ factor comes from Merkle proofs within the PMR tree. While asymptotically minor, it adds implementation complexity.
5. **Not Optimized for Batch Proofs:** Multiple independent proofs don't naturally share structure.

8.3 Comparative Analysis: Deep Insights

8.3.1 The Fundamental Tension

KIP-15 and KIP-6 represent two orthogonal responses to the same underlying challenge: how to provide verifiable historical information in a pruning blockchain.

- **KIP-15** prioritizes *ordering integrity*—ensuring the canonical sequence of transactions is cryptographically committed and verifiable.
- **KIP-6** prioritizes *proof efficiency*—ensuring historical claims can be verified with logarithmic-sized proofs.

Neither alone solves the complete problem. A Layer-2 rollup or cross-chain bridge needs *both*: proof that a transaction was accepted (KIP-6's domain) *and* proof of its position in the global order (KIP-15's domain).

8.3.2 The Architectural Incompatibility

Implementing KIP-15 and KIP-6 together creates architectural friction:

1. **Dual Commitment Overhead:** Two separate header fields, two separate validation paths, two separate proof structures.
2. **Proof Composition Complexity:** A complete "transaction receipt" would require:
   - A PoChM (from KIP-6) proving block chain membership
   - A Merkle proof for transaction within block's ATMR
   - A chain of SequencingCommitments (from KIP-15) if ordering matters

   The third component reintroduces linear complexity, negating KIP-6's benefits for order-sensitive applications.
3. **Conceptual Overlap Without Synergy:** Both proposals touch block headers and validation rules, but don't reinforce each other's proofs.

8.3.3 Why MMR is the Natural Synthesis

Our MMR-based approach resolves this tension through *structural unification*:

1. **Inherent Ordering via Leaf Sequence:** Each MMR leaf corresponds to a specific position in the selected chain. The leaf's index *is* the block's position in the canonical sequence. Transaction order is encoded by: (a) the block's MMR leaf index, and (b) the transaction's position within the block's OrderRoot.
2. **Logarithmic Proofs for Both Membership and Order:** A single MMR proof demonstrates both that a block was in the chain *and* its exact position. No linear traversal is needed.
3. **Unified Header Field:** The `order_root` commits to transaction ordering (KIP-15's goal), while the `mmr_root` enables efficient proofs (KIP-6's goal). These naturally compose rather than compete.

8.3.4 Quantitative Analysis: Gas Cost Comparison

For cross-chain bridge applications, on-chain verification cost is critical. We compare the gas requirements for verifying a transaction that occurred D blocks ago:

Proposition 8.2 (Gas Cost Bounds). *For a proof of depth D blocks:*

$$Gas_{KIP\text{-}15} = 1920 \times D \tag{9}$$

$$Gas_{KIP\text{-}6} \approx 1920 \times (2\log_2 D \times \lceil \log_2 \log_2 D \rceil + 1) \tag{10}$$

$$Gas_{MMR} \approx 1920 \times (\log_2 D + k) \tag{11}$$

where k is the number of peaks ($k \leq \log_2 D$, typically much smaller).

At $D = 100,000$ blocks ($\sim$2.8 hours at 10 BPS):

- KIP-15: 192 million gas (exceeds block limit by 6×)
- KIP-6: $\sim$104,000 gas (feasible)
- MMR: $\sim$61,000 gas (feasible, 41% lower than KIP-6)

Table 6: Detailed Feature Comparison: KIP-6, KIP-15, and MMR-Based Approach

Aspect	**KIP-15**	**KIP-6**	**MMR (Ours)**
Core Functionality			
Commits TX Order	✓ (chain hash)	✗	✓ (via OrderRoot)
Efficient Inclusion Proof	✗ ($\mathcal{O}(N)$)	✓ ($\mathcal{O}(\log N \log\log N)$)	✓ ($\mathcal{O}(\log N)$)
Proves Position in History	✓ (implicitly)	✗ (block only)	✓ (explicitly)
Proof Characteristics			
Proof size (depth D)	$\mathcal{O}(D) \times 32$B	$\mathcal{O}(\log D \log\log D) \times 32$B	$\mathcal{O}(\log D) \times 32$B
Proof at 100K blocks	∼3.2 MB	∼9 KB	∼1 KB
Proof at 1M blocks	∼32 MB	∼11 KB	∼1.1 KB
On-chain verifiable	✗ (exceeds gas)	✓	✓
Implementation			
Header overhead	0 bytes (replace)	+32 bytes	+40 bytes (net)
Hashes per block	1	∼33 (chain blocks)	∼5–10 (average)
External dependency	None	Posterity headers	None
Use Case Support			
ATAN sync (ordering)	✓ (linear)	✗	✓ (logarithmic)
L2 rollup proofs	✗ (too large)	Partial (no order)	✓
Cross-chain bridge	✗	✓	✓
Light client SPV	✗	✓	✓
Batch proof optimization	✗	Limited	✓ (shared peaks)

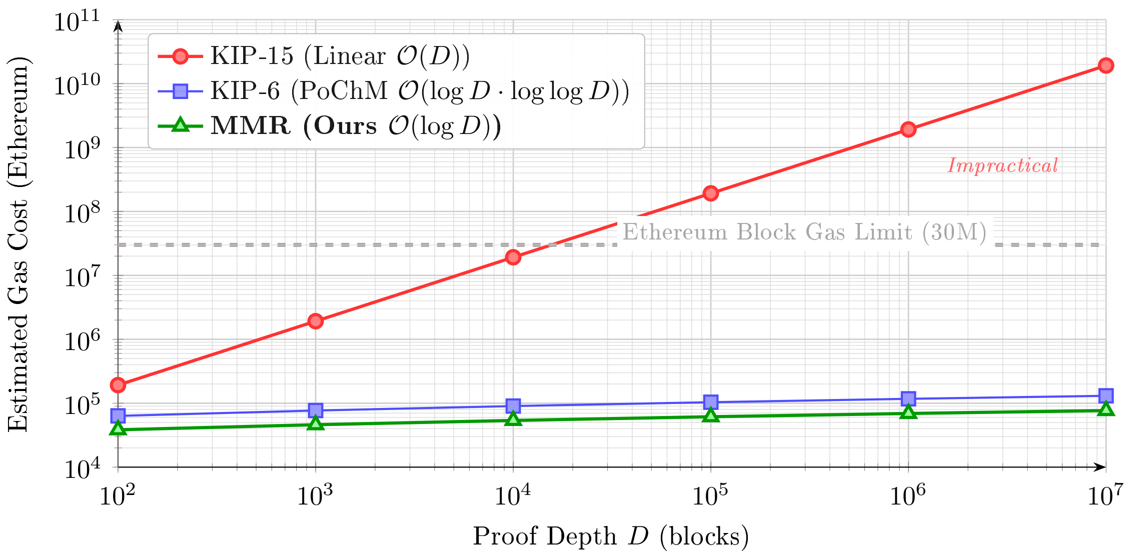


Figure 11: On-chain verification gas cost comparison. KIP-15's linear growth makes it impractical beyond ∼15,000 blocks (crosses gas limit). Both KIP-6 and MMR remain feasible at any depth, with MMR offering ∼40% lower costs due to simpler proof structure.

8.3.5 Architectural Elegance: The MMR Advantage

Beyond quantitative metrics, the MMR approach offers qualitative advantages:

1. **Conceptual Simplicity:** One data structure serves both purposes. Developers need only understand MMRs, not the interaction between skip-lists (KIP-6) and hash chains (KIP-15).
2. **Proof Composability:** Multiple transactions from different blocks can share the same peak list in their proofs, enabling efficient batch verification.
3. **No Posterity Dependency:** Unlike KIP-6, MMR proofs can be generated immediately after block confirmation—no waiting for posterity header creation.
4. **Natural Append-Only Semantics:** MMR's structure mirrors the blockchain's append-only nature, making reorg handling straightforward.
5. **Industry Precedent:** MMRs are battle-tested in Grin, Beam, and Nervos, providing implementation guidance and security confidence.

8.4 Summary: Why Unification Matters

The fundamental insight driving our proposal is that *ordering commitment and efficient proofs are not independent problems*—they are two facets of the same challenge: providing verifiable historical information in a resource-constrained environment.

KIP-15 asks: "How do we commit to the order?"
KIP-6 asks: "How do we prove membership efficiently?"
MMR answers both: "By embedding the order in an efficient accumulator."

Implementing KIP-6 and KIP-15 separately would require:

- Two consensus changes
- Two header fields
- Two validation codepaths
- Two proof formats
- Complex interaction semantics

Our MMR-based design delivers equivalent (and superior) functionality through a single, unified mechanism—embodying the engineering principle that the best solution is often one that *eliminates* the distinction between seemingly separate problems.

This aligns with broader industry trends: Ethereum's Verkle trees for state, Bitcoin's work on Utreexo, and various succinct proof protocols all strive to make blockchain data more accessible without full trust or full nodes. Kaspa's adoption of MMR commitments would be among the first in a live L1 blockchain, and we believe it will greatly enhance the ecosystem's capabilities.

9 Implementation and Deployment

9.1 Activation via Network Upgrade

Because this change affects the consensus (block header and validation), it must be introduced via a **hard fork**. We choose an activation point (e.g., a DAA score threshold or timestamp). Blocks after that must include the new fields and follow the new validation logic.

Table 7: Summary Comparison of Transaction Commitment Approaches

Capability	**KIP-15**	**KIP-6**	**MMR (Ours)**
Commits to global order	✓	✗	✓
Supports $\mathcal{O}(\log n)$ proofs	✗	✓	✓
Jump/skip verification	✗	✓	✓
L2-friendly (gas cost)	Poor	Good	Excellent
Additional data structures	None	Complex	None
Implementation complexity	Low	Medium/High	Medium

9.2 Miner Integration

Miners need to upgrade their block template generation:

1. Compute `order_root` from accepted transactions
2. Fetch current MMR state from node (peaks and size)
3. Compute new leaf hash and updated `mmr_root` and `mmr_size`
4. Populate these in the header, then perform PoW

9.3 Data Structures and APIs

New database structures:

- `mmr_state_store`: Block hash → MMR state (peaks list + size)
- `mmr_node_store`: MMR position → hash (optional, for proof generation)

New RPC endpoints:

- `GetMmrProof(blockHash, targetBlockHash)`: Returns inclusion proof
- `VerifyTransactionProof(proof)`: Validates a given proof

9.4 Security Considerations

- **Collision Resistance:** Domain separation ensures no hash collisions across different contexts
- **Replay Protection:** MMR root is in the block header, so tampering requires re-mining
- **Denial of Service:** Proof verification is fast (~30 hashes maximum)
- **Reorg Exploit:** Cryptographic hashing prevents same MMR root for different histories

10 Conclusion

We have presented a unified approach to incorporating Sequencing Commitments in Kaspa using a Merkle Mountain Range (MMR) structure, achieving both canonical transaction ordering commitment and efficient logarithmic proofs for transaction inclusion. This design meets the requirements of KIP-15 and KIP-6 simultaneously, simplifying implementation and maintenance.

10.1 Summary of Results

Our scheme adds a small overhead (40 bytes per block) to the Kaspa headers, which is negligible relative to block sizes and bandwidth. It modestly increases storage requirements (on the order of tens of GB per year for a full archival node), which is a worthwhile trade for the added functionality. In return, Kaspa gains:

1. **Verifiable Transaction Order:** Every block locks in the exact order of all transactions up to that point. This enhances the determinism and auditability of the ledger.

2. **Compact Proofs:** Any transaction in history can be proven to a light client or contract with a proof on the order of 0.5–1 KB, regardless of how old it is, enabling cross-chain verification and light clients without trusted servers.

3. **Support for Decentralized L2:** Rollups on Kaspa can use the L1 as a data availability and ordering layer directly, leveraging the commitment to sequence. L2 state transitions can be verified knowing the order of L1 transactions (no alternative histories to consider).

4. **Efficient Sync & Recovery:** Nodes can sync faster and recover from data loss using the MMR commitments to validate bulk data from untrusted sources. This reduces reliance on trusted archives or intensive bootstrap processes, contributing to network resilience and decentralization.

5. **Self-Authenticating History:** The whole history becomes verifiable against the latest block. Any single block's wrong data breaks the MMR root instantly—analogous to how a Merkle root turns a block's transactions into a verifiable set, now extended to the entire blockchain level.

The unified approach subsumes both KIP-15 (sequencing) and KIP-6 (logarithmic proofs) in a single coherent design, simplifying implementation while delivering enhanced functionality. The proof size remains around 1 KB even for millions of blocks, and on-chain verification requires only $\sim$30 hashes ($\sim$60,000 gas on Ethereum).

10.2 Future Work

- **SNARK/STARK Compression:** Potentially reduce proof size to constant using zero-knowledge proofs, enabling even more efficient cross-chain verification

- **Multi-proof Aggregation:** Batch verification optimizations for bridge contracts handling many transactions simultaneously

- **Cross-chain Applications:** Adapting the scheme for other DAG-based ledgers or even traditional blockchains

- **Light Client Protocols:** Building improved light client protocols where light nodes only track headers and use the MMR for occasional proofs to confirm state changes or detect forks—essentially enabling NiPoPoW-like proofs of transactions, not just chain progress

- **Transaction Exclusion Proofs:** Ensuring a transaction was not included by a certain point (can be achieved by proving inclusion of a conflicting tx or reaching a certain block without inclusion)

- **Security Analysis:** Formal security analysis proving that subverting the log (e.g., including a transaction out of order or forging a history) would require breaking cryptographic hash collision resistance

Our approach underscores a general principle: by turning the blockchain (or blockDAG) history into a cryptographically authenticated data structure (a verifiable log), we empower light clients and cross-chain systems to interact with the blockchain trustlessly. As Kaspa continues to push the limits of blockDAG performance, features like this ensure that its security and usability for advanced applications keep pace. We anticipate this will accelerate development of L2 ecosystems around Kaspa and serve as a model for other projects seeking efficient historical proof systems.

References

[1] Kaspa Improvement Proposal KIP-15: Sequencing Commitments for Transaction Ordering. *Kaspa Updates to Crescendo and 10BPS*, 2024. `https://kaspa.org/kaspa-updates-to-crescendo-and-10bps/`

[2] Kaspa Improvement Proposal KIP-6: Logarithmic Transaction Inclusion Proofs. *Kaspa Improvement Proposals*, 2023. `https://github.com/kaspanet/kips/blob/master/kip-0006.md`

[3] Herodotus Protocol Documentation: Merkle Mountain Ranges. *Herodotus Docs*, 2024. `https://docs.herodotus.dev/herodotus-docs/protocol-design/historical-block-hash-accumulator/merkle-mountain-ranges`

[4] Y. Sompolinsky, S. Wyborski, and A. Zohar, "Phantom and GhostDAG: A Scalable Generalization of Nakamoto Consensus," *Cryptology ePrint Archive*, Report 2018/104, 2021.

[5] Nervos Network, "RFC: Merkle Mountain Range," *Nervos RFCs*, 2019.

[6] Grin Documentation, "Merkle Mountain Ranges," *Grin Wiki*, 2019.

[7] P. Todd, "Merkle Mountain Ranges," *Bitcoin Development Mailing List*, 2012.

[8] M. Sompolinsky et al., "PHANTOM GHOSTDAG: A Scalable Generalization of Nakamoto Consensus," *Proceedings of the 3rd ACM Conference on Advances in Financial Technologies*, 2021.

[9] G. Wood, "Ethereum: A Secure Decentralised Generalised Transaction Ledger," *Ethereum Yellow Paper*, 2016.

[10] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008.

[11] B. Bünz, L. Kiffer, L. Luu, and M. Zamani, "FlyClient: Super-Light Clients for Cryptocurrencies," *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[12] A. Kiayias, A. Miller, and D. Zindros, "Non-Interactive Proofs of Proof-of-Work," *Financial Cryptography and Data Security*, 2020.

[13] G. A. Marson, et al., "The Merkle Mountain Belt," *arXiv preprint arXiv:2511.13582*, 2025.

[14] Chainalysis, "Crypto Bridge Hacks in 2022: Statistics on Bridge Vulnerabilities," *Chainalysis Reports*, 2022.