Logarithmic Sequencing Commitments for GHOSTDAG BlockDAGs
via Merkle Mountain Ranges
Consensus-Level Execution Order Proofs with $\mathcal{O}(\log n)$ Verification

Arthur Zhang[*]
Tondi Foundation

Neo Maxwell
Tondi Foundation

January 2026

Abstract

We propose a Merkle Mountain Range (MMR) based *Execution Log* scheme for GHOSTDAG-style blockDAG protocols that output a deterministic total order of accepted transactions. Unlike auxiliary history logs, our approach treats the MMR as a **first-class consensus artifact**—the `exec_root` (Merkle root of accepted transaction IDs) and `elog_root` (execution log accumulator root) are *outputs of consensus execution itself*, validated alongside UTXO commitments. A block with incorrect execution log fields is invalid at the consensus level, not merely inconsistent with an external index.

We formalize the **Consensus API Contract**: GHOSTDAG implementations must expose a deterministic `AcceptedTxList(B)` for each block B, representing the exact sequence of transaction IDs accepted during UTXO validation. The `exec_root` commits to this sequence, and the MMR leaf binds `exec_root` to the block's selected parent hash, creating an append-only execution log. This design ensures that the commitment reflects the *actual execution order*—not merely block ordering. Layer-2 systems can prove that a transaction was executed at position (j, i): block j in the selected chain, transaction i within that block's accepted list.

We analyze the overhead based on experiments with Tondi, a GHOSTDAG implementation running at 10 blocks per second. The scheme adds 40 bytes per header (negligible bandwidth, $< 0.1\%$) and moderate storage (tens of GB/year for archive nodes). In return, the system gains $\mathcal{O}(\log n)$ execution proofs: verifying a transaction from 100,000 blocks ago requires $\sim$17 hashes ($<$1 KB), versus 100,000 hashes in linear schemes. We present use cases including trustless cross-chain bridges, fast node bootstrapping via skip proofs, and verifiable disaster recovery.

Keywords: Merkle Mountain Range, GHOSTDAG, Execution Log, Consensus Commitment, Logarithmic Proofs, Layer-2 Scaling, 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.

Terminology: We distinguish between *BlockDAG* as a structural paradigm (blocks forming a DAG rather than a chain, allowing parallel block production) and *GHOSTDAG* as a specific consensus algorithm that operates on BlockDAGs. GHOSTDAG provides rules for ordering blocks (via blue scores), selecting a canonical chain (the selected parent chain), and merging parallel blocks into a deterministic execution sequence. Our scheme applies specifically to **GHOSTDAG-style protocols**—those that output a deterministic total order of accepted transactions via selected-parent-chain + mergeset ordering.

GHOSTDAG-based blockDAGs are high-throughput protocols where blocks are produced in parallel (multi-leader consensus). Rather than a single chain of blocks, GHOSTDAG 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.
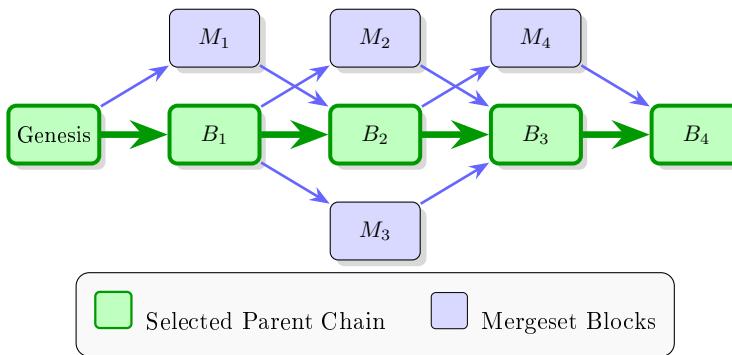

Figure 1: GHOSTDAG BlockDAG Structure. The green blocks form the *selected parent chain* (the canonical backbone determined by GHOSTDAG). Blue blocks are *mergeset blocks*—parallel blocks whose transactions are merged into the ledger when a selected-chain block references them as additional parents.

Two key challenges arise in GHOSTDAG-based blockDAGs:

Sequencing Commitment: 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. A straightforward approach is 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: $SC_n = H(SC_{n-1} \parallel ExecRoot_n)$. While simple, 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: 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. Naive proofs of inclusion in any blockDAG could be linear in the distance from the block to be verified. The challenge is to use a data structure that enables skip-ahead verification, reminiscent of the FlyClient approach for light clients [11].

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 From Block Ordering to Execution Ordering

A critical insight motivating our design is the distinction between *block ordering* and *execution ordering*. Traditional approaches focus on committing to the order of blocks. However, what Layer-2 systems, bridges, and rollups actually need is the **execution order of transactions**—the deterministic sequence in which transactions are validated and applied to the UTXO set.

In a GHOSTDAG structure, a single block may "merge" transactions from multiple parallel blocks in its mergeset. The execution order is determined not by block arrival time, but by the consensus algorithm's deterministic processing of the selected parent chain. A transaction's position in the global execution order is a function of:

1. Which selected-chain block processes it

2. Its position within that block's accepted transaction list (derived from UTXO validation)

This distinction matters because:

- An L2 rollup must replay transactions in *exactly* the same order to reproduce state

- A bridge contract verifying withdrawals needs to confirm *execution position*, not just block inclusion

- Dispute resolution requires proving that a transaction was executed at a specific global position

Our approach makes this execution order a **first-class consensus output**: the `exec_root` is computed directly from the `accepted_tx_ids` produced by the UTXO validation pipeline, not from block metadata.

1.2 Our Contributions

Our approach, which we call the *Execution-MMR Sequencing Commitment*, integrates with GHOSTDAG consensus as follows:

- Every block contains an `exec_root` computed as MerkleRoot(accepted_tx_ids), where `accepted_tx_ids` is the deterministic output of the consensus execution pipeline (`calculate_utxo_state`). This is not a post-processing step—it is derived from the same computation that determines UTXO validity.

- Every block also carries an `elog_root` and `elog_size`, which accumulate all prior blocks' execution commitments in an append-only Merkle Mountain Range. The MMR leaf binds the block's `exec_root` to its position in the selected parent chain.

- The combination allows any transaction's **execution position** to be proven with two linked proofs: an intra-block Merkle proof showing the transaction's index within the block's execution order, and an MMR proof showing that block's position in the global execution history.

This design effectively provides **everlasting cryptographic execution receipts** for transactions: a light client or an L2 contract can verify that a transaction was executed at position (j, i)—block j in the selected chain, transaction i within that block—by checking a short proof against a recent block header's execution log 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. **Execution-Level Commitment Scheme:** We design a consensus change that introduces `exec_root` and `elog_root` as *consensus execution outputs*. Unlike auxiliary logs, these commitments are validated as part of block acceptance—a wrong `exec_root` or `elog_root` causes block rejection, just like a wrong UTXO commitment.

2. **Formalization of Canonical Execution Order:** We define TxList(B) as the exact sequence of accepted transaction IDs produced by `calculate_utxo_state()`. This is not a specification we impose—it is the existing deterministic behavior of GHOSTDAG consensus. We simply make it externally verifiable.

3. **Unified Ordering and Proof Mechanism:** The `exec_root` achieves the ordering commitment goal. The `elog_root` achieves the logarithmic proof goal. Together, they provide a single, unified mechanism for both transaction ordering and efficient historical proofs.

4. **Two-Layer Proof Model:** We formalize the proof structure as:

   - **Intra-block execution proof:** Transaction $\rightarrow$ `exec_root` (proves execution index within block)
   - **Inter-block MMR proof:** Block leaf $\rightarrow$ `elog_root` (proves block's position in execution history)

   Together, these prove the transaction's global execution position with $\mathcal{O}(\log n)$ complexity.

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

1.3 Paper Organization

The rest of this paper is organized as follows. Section 2 provides background on GHOSTDAG 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 based on experiments with Tondi, a GHOSTDAG implementation running at 10 BPS. Section 9 compares our approach with alternative schemes. Implementation considerations are given in Section 10. Finally, Section 11 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).

The blockDAG architecture complicates transaction ordering in two ways:

1. There isn't a single chain of blocks—blocks can be created concurrently and reference multiple parents. 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. GHOSTDAG 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}.*

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

MMR with 13 leaves: $13 = 8 + 4 + 1 = 2^3 + 2^2 + 2^0$
Three peaks are "bagged" (hashed
together) into a single root
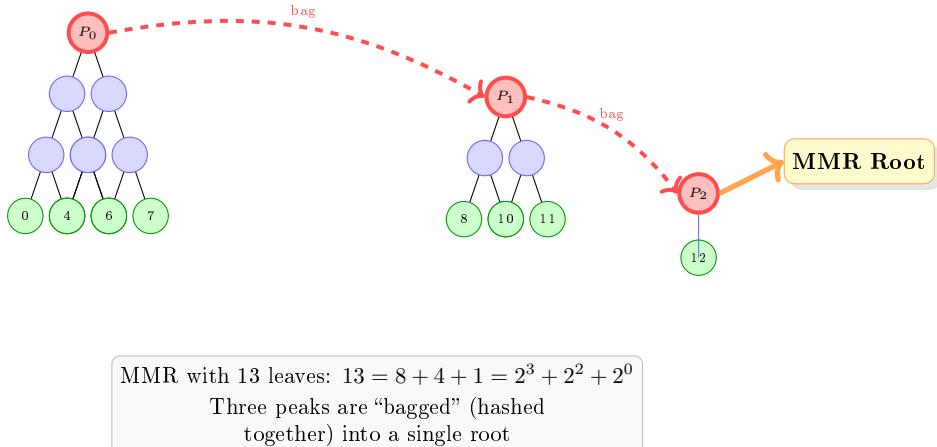
Figure 2: Merkle Mountain Range structure with 13 leaves. The structure consists of three "mountains" (perfect binary trees) corresponding to the binary representation $13_{10} = 1101_2$. Peaks P_0, P_1, P_2 are hashed together (bagged) to produce the final MMR root.

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 GHOSTDAG, Selected Chain, and Consensus-Determined Transaction Sequence

GHOSTDAG BlockDAG: A GHOSTDAG-based blockchain is a blockDAG where blocks reference multiple parents. This yields high throughput (e.g., 10 blocks per second in Tondi) but requires a rule to determine a total ordering. 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*.

2.3.1 Formal Model: Transaction Sequence Derivation

We now formalize the exact sequence that our scheme commits to. This is critical: reviewers and implementers must understand *precisely* which sequence is being committed.

Definition 2.2 (GHOSTDAG Block Ordering). *For a block B on the selected parent chain, define:*

- $\mathsf{SP}(B)$: *The selected parent of B*

- $\mathsf{Mergeset}(B)$: *The set of blocks referenced by B but not on the selected parent chain*

- $\mathsf{MergesetOrder}(B)$: *A deterministic total ordering of $\mathsf{Mergeset}(B)$, typically by $(blue\_score, block\_hash)$ lexicographically*

Definition 2.3 (Intra-Block Transaction Ordering). *For each block M, define $\mathsf{TxOrder}(M)$ as the sequence of transactions in M's body, ordered by their position in the block's transaction list (as serialized in the block). This ordering is **miner-chosen** but **consensus-fixed** once the block is created.*

Definition 2.4 (Conflict Resolution Rule). *When processing transactions, if transaction tx_a and tx_b conflict (e.g., double-spend the same UTXO):*

1. *If they appear in different blocks: the transaction in the* earlier *block (per $\mathsf{MergesetOrder}$) wins*

2. *If they appear in the same block: the transaction with* lower index *(per $\mathsf{TxOrder}$) wins*

*The losing transaction is **rejected** and does not appear in $\mathsf{AcceptedTxList}$.*

Definition 2.5 (Canonical Execution Sequence). *For a block B on the selected parent chain, the* canonical execution sequence *$\mathsf{TxList}(B)$ is constructed as follows:*

1. *Let $\mathsf{BlockSeq}(B) = [\mathsf{SP}(B)] \oplus \mathsf{MergesetOrder}(B) \oplus [B]$*

2. *For each block $M \in \mathsf{BlockSeq}(B)$ in order:*

   (a) *For each $tx \in \mathsf{TxOrder}(M)$ in order:*

      i. *If tx is valid against current UTXO state and not conflicting with already-accepted transactions: accept tx, append $tx.id$ to $\mathsf{TxList}(B)$, update UTXO state*

      ii. *Otherwise: reject tx (do not append)*

   *This produces a **deterministic, ordered list** of accepted transaction IDs.*

Proposition 2.6 (Consensus Uniqueness). *For any block B and any two honest nodes N_1, N_2:*

$$\mathsf{TxList}_{N_1}(B) = \mathsf{TxList}_{N_2}(B)$$

provided both nodes have the same view of the DAG up to B.

Proof Sketch: The sequence depends only on: (1) the DAG structure (which blocks exist and their parent relationships), (2) the GHOSTDAG blue score assignments (deterministic from DAG), (3) the block body contents (transactions and their ordering). All three are consensus-determined and identical across honest nodes. $\square$

Key Insight: The miner of block B controls $\mathsf{TxOrder}(B)$ (the ordering of B's own transactions), but *not* the ordering of mergeset blocks or the conflict resolution outcome. Thus, while miners have some influence over ordering, the *overall sequence across the selected chain* is consensus-determined, not miner-arbitrary.

Definition 2.7 (Execution Root). *For a block B:*

$$\mathsf{ExecRoot}(B) = \mathsf{MerkleRoot}(\mathsf{TxList}(B))$$

where $\mathsf{MerkleRoot}$ is a standard binary Merkle tree over the ordered list of transaction IDs.

This formalization ensures that $\mathsf{ExecRoot}(B)$ commits to a **precisely defined, consensus-unique sequence**—not an implementation-dependent artifact.

2.3.2 Consensus API Contract

For our execution log scheme to be implementable, GHOSTDAG implementations MUST expose a specific interface. We formalize this as a **Consensus API Contract**:

Definition 2.8 (Consensus API Contract). *A GHOSTDAG implementation supporting the execution log MUST provide the following deterministic function:*

$$AcceptedTxList(B) \rightarrow [tx_1, tx_2, \ldots, tx_k] \tag{1}$$

*where the output is the **ordered list** of transaction IDs accepted when validating block B. This function MUST satisfy:*

1. ***Determinism:*** *For any block B and parent state S, all honest nodes produce identical output*

2. ***Atomicity:*** *The list is produced atomically with UTXO state transition*

3. ***Totality:*** *Every accepted transaction appears exactly once in the list*

4. ***Conflict Resolution:*** *Double-spends are resolved before output; only winners appear*

Interface Binding: In practice, this maps directly to existing GHOSTDAG implementations. For example, in a typical implementation:

```
def calculate_utxo_state(block, parent_state):
    ctx = UTXOContext(parent_state)
    # Process selected parent, mergeset, and block's own txs
    for tx in ordered_transactions(block):
        if ctx.validate_and_apply(tx):
            ctx.accepted_tx_ids.append(tx.id)
    return ctx  # ctx.accepted_tx_ids is AcceptedTxList(B)
```

Listing 1: Consensus API Contract binding
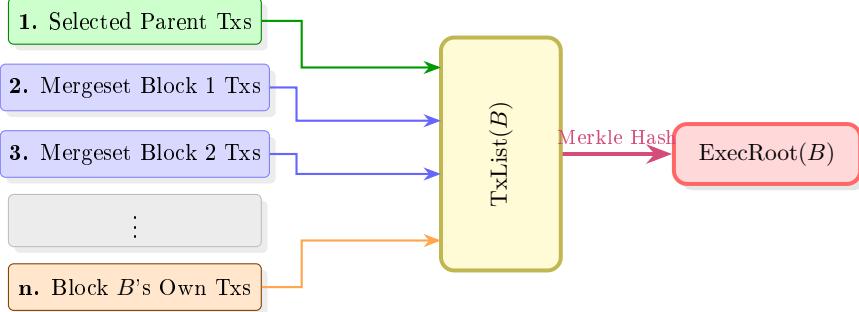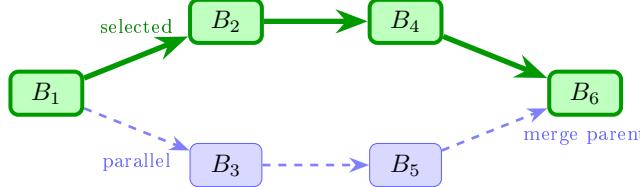
Figure 3: Construction of the canonical transaction list (TxList) and ExecRoot for block B. Transactions are ordered deterministically: selected parent first, then mergeset blocks in consensus order, finally block B's own transactions.

Reorg Semantics: When a reorganization occurs (selected chain changes), the `AcceptedTxList` for affected blocks MUST be recomputed. The execution log tracks the *current* selected chain view; historical proofs become invalid if the chain reorganizes past their anchor point. We formalize this in Section 4.2.

A GHOSTDAG 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: $B_1 \rightarrow B_2 \rightarrow B_4 \rightarrow B_6$ (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), GHOSTDAG consensus ensures only one will be accepted—generally the one appearing in the earlier position in the canonical order.

A linear sequencing mechanism would maintain a running hash chain: $SC_n = H(SC_{n-1} \parallel \text{ExecRoot}(B_n))$. 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.

11

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 GHOSTDAG 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 GHOSTDAG block header with two new fields and modify one existing field:

Table 1: Block Header Field Modifications

Field	Size	Description
`exec_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
`elog_root`	32 bytes	MMR root hash after inserting this block's leaf, serving as a cryptographic commitment to the entire history of transaction order
`elog_size`	8 bytes	Total number of leaves in the MMR after adding this block
Removed field:		
`accepted_id_merkle_root`	32 bytes	(Replaced by `exec_root`)
Net Change	**+40 bytes**	($2 \times 32 + 8 - 32 = 40$ bytes per header)

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

3.2 Consensus Impact Statement

We explicitly characterize the consensus-level changes required by this proposal:

Table 2: Consensus Impact Summary

Aspect	Impact	Details
Block hash computation	**Changed**	Block hash now includes `exec_root`, `elog_root`, `elog_size`
Validation rules	**Extended**	Nodes must validate new fields (see below)
PoW security	**Unchanged**	PoW target and difficulty adjustment unaffected
Fork choice rule	**Unchanged**	GHOSTDAG blue score / selected parent rules unchanged
Activation	**Hard fork**	Requires coordinated upgrade at designated height

3.2.1 New Validation Rules

Each validating node MUST perform the following checks when receiving a new block B:

1. **ExecRoot Validation:**

(a) Execute `calculate_utxo_state(B)` to obtain `ctx.accepted_tx_ids`

(b) Compute expected_exec_root = MerkleRoot(ctx.accepted_tx_ids)

(c) REJECT block if B.exec_root $\neq$ expected_exec_root

2. **ElogRoot Validation:**

(a) Retrieve MMR state from B's selected parent: mmr_state = GetMmrState(SP(B))

(b) Compute leaf: $L = H_{\text{leaf}}(\text{DOMAIN\_LEAF}\|\text{hash(SP}(B))\|B.\text{exec\_root})$

(c) Append leaf: (new_state, expected_root) = mmr_state.append(L)

(d) REJECT block if B.elog_root $\neq$ expected_root

(e) REJECT block if B.elog_size $\neq$ new_state.size

3.2.2 Impact on Block Hash

The block header hash (used for PoW and block identification) MUST include the new fields. This ensures:

- A miner cannot change execution log fields without re-mining

- The execution log commitment is protected by PoW security

- Block identity is bound to execution history commitment

3.2.3 Reorg Handling

When the selected parent chain changes (reorganization):

1. Identify the common ancestor C of old and new chains

2. For each block from C to new tip: recompute MMR state (peaks and size)

3. **Invariant:** MMR state is stored per-block, not just at virtual tip

This is $\mathcal{O}(k)$ where k is the reorg depth, typically small.

3.2.4 Activation and Upgrade Path

- **Activation height:** A predetermined block height/score at which validation rules become enforced

- **Genesis initialization:** The genesis block has elog_root = $\emptyset$ (empty MMR), elog_size = 0

- **Backward compatibility:** Pre-activation blocks do not have execution log fields; post-activation nodes must not validate these fields for old blocks

- **Upgrade requirement:** All validating nodes must upgrade before activation height (hard fork)

3.3 Canonical Execution Order and ExecRoot Computation

Definition 3.1 (Canonical Transaction List). *For each block B, we define an ordered list of transaction IDs, denoted* $\text{TxList}(B)$*, which is the sequence of all accepted transactions **as determined by the consensus execution pipeline**. Critically, this is not a separate computation—it is the direct output of* `calculate_utxo_state()`*, specifically the* `accepted_tx_ids` *field of the* `UtxoProcessingContext`*.*

The `calculate_utxo_state()` function processes:

1. The coinbase transaction of the selected parent P_{sel} (position 0)

2. Other transactions from P_{sel}, in their original block order, validated against UTXO

3. Transactions from mergeset blocks, ordered by `consensus_ordered_mergeset_without_selected_parent`, each block's transactions in original order

4. Each transaction is validated; conflicts are deterministically resolved (first valid wins)

Key Design Principle: We do *not* recompute or reorder transactions for the commitment. We directly use the `accepted_tx_ids` output from consensus:

$$\text{TxList}(B) := \text{ctx.accepted\_tx\_ids} \quad (\text{from } \texttt{UtxoProcessingContext}) \tag{2}$$

$$\text{ExecRoot}(B) := \text{MerkleRoot}(\text{TxList}(B)) \tag{3}$$

Consensus Integration: The `exec_root` is validated as part of block acceptance. If the miner's claimed `exec_root` does not match $\text{MerkleRoot}(\text{accepted\_tx\_ids})$, the block is rejected with `BadExecRoot`. This makes the execution order commitment a *consensus-level* requirement, not an auxiliary log.

```
1  def compute_exec_root(ctx):
2      # ctx.accepted_tx_ids is the deterministic output
3      # of calculate_utxo_state() - we use it directly
4      return merkle_root(ctx.accepted_tx_ids)
```
<div align="center">Listing 2: ExecRoot Computation (Pseudocode)</div>

Why This Matters: By deriving `exec_root` from the same computation that determines UTXO validity, we ensure that:

- A wrong `exec_root` = consensus violation = block rejected

- The commitment reflects *actual execution*, not just declared block contents

- L2 systems can trust that the order matches what would be executed

3.4 MMR Leaf and Internal Hashing Scheme

For each block B that lies on the selected parent chain, we create an MMR leaf node that binds the block's **execution output** to its **chain position**:

$$L(B) = \text{H}_{\text{leaf}}(\text{DOMAIN\_LEAF} \parallel h_{P_{\text{sel}}(B)} \parallel \text{ExecRoot}(B)) \tag{4}$$

where:

- DOMAIN_LEAF is a fixed domain separation prefix (**"ELOG_LEAF_V1"**)

- $h_{P_{\text{sel}}(B)}$ is the hash of B's selected parent block header

- ExecRoot(B) is the Merkle root of accepted transactions (execution order commitment)

Design Rationale (Three Critical Properties):

1. **Avoids Self-Hash Circular Dependency:** We do *not* include the block's own hash h_B in the leaf. This is essential because $h_B = H(\text{header})$ and the header contains `elog_root`, which depends on the leaf. Including h_B would create an unsolvable circular dependency.

2. **Binds MMR to Execution Order:** By including ExecRoot(B), the MMR leaf commits to the *execution output* of block B, not just its existence. A change in execution order (different `accepted_tx_ids`) changes the leaf hash, which changes the MMR root.

3. **Enables Proof-Execution Alignment:** The proof verifier can reconstruct the leaf from $(h_{P_{\text{sel}}}, \text{ExecRoot})$—both of which are available from block headers. This allows verification without access to block bodies.

Why Not Include `tx_count` or `daa_score`?

- `tx_count`: Already implicit in ExecRoot (the Merkle root structure encodes tree shape)

- `daa_score`: Redundant—$h_{P_{\text{sel}}}$ uniquely identifies chain position

- Minimizing leaf contents simplifies proofs without losing security

Leaf Uniqueness: The pair $(h_{P_{\text{sel}}(B)}, \text{ExecRoot}(B))$ uniquely identifies a block's execution commitment. Two blocks with the same selected parent but different accepted transactions produce different leaves. Two blocks with different parents produce different leaves regardless of transactions.

3.4.1 Hash Function Specification

For reproducibility, we specify all hash computations precisely:

Table 3: Domain-Separated Hash Specifications

Context	Domain Prefix	Input Format
Leaf	`ELOG_LEAF_V1`	prefix$\|h_{\text{parent}}\|$exec_root
Internal Node	`ELOG_NODE_V1`	prefix$\|h_{\text{left}}\|h_{\text{right}}$
Peak Bagging	`ELOG_BAG_V1`	prefix$\|h_{\text{acc}}\|h_{\text{peak}}$
MMR Root	`ELOG_ROOT_V1`	prefix$\|h_{\text{bag}}\|$size_le64
ExecRoot	(standard Merkle)	Binary Merkle tree over $[\text{tx\_id}_1, \ldots, \text{tx\_id}_k]$

Encoding Rules:

- **Hash function:** BLAKE3-256 (or SHA-256 for compatibility)

- **Domain prefixes:** ASCII-encoded, null-terminated, padded to 16 bytes

- **Hash values:** 32 bytes, big-endian

- **Size encoding:** 8 bytes, little-endian unsigned 64-bit integer

- **Concatenation:** Direct byte concatenation, no length prefixes

Example (Leaf Hash):

```
1  def compute_leaf(parent_hash: bytes, exec_root: bytes) -> bytes:
2      domain = b"ELOG_LEAF_V1\x00\x00\x00\x00"  # 16 bytes
3      preimage = domain + parent_hash + exec_root  # 16 + 32 + 32 = 80 bytes
4      return blake3(preimage)
```

Listing 3: Leaf hash computation example

This specification ensures that independent implementations produce identical results, enabling cross-client verification.
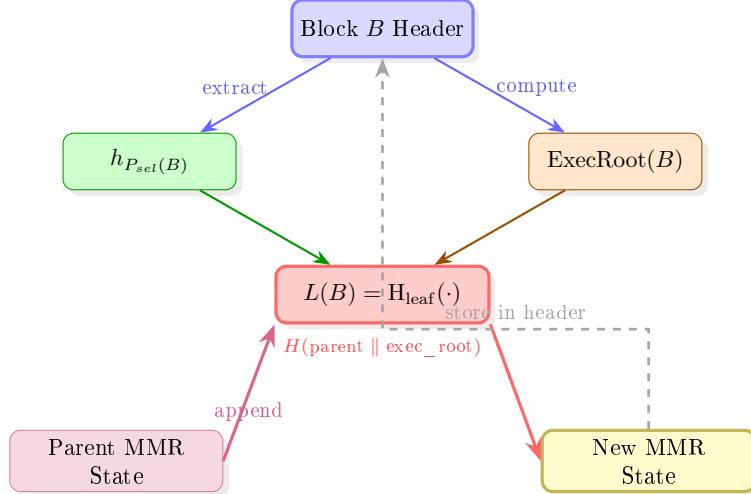
Figure 4: MMR leaf computation and state update flow. The leaf is computed from parent hash and ExecRoot, then appended to the parent's MMR state to produce the new MMR root stored in the block header.

MMR Structure Maintenance: The MMR is maintained as follows. Let $S(n)$ denote the state of the MMR after processing n leaves. The initial state $S(0)$ has an empty peaks list and size 0:

$$S(0) = \text{MmrState}\{\text{peaks} : [], \text{size} : 0\} \tag{5}$$

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

Genesis Block: The genesis block's parent MMR state is $S(0)$. After processing the genesis block, the MMR contains exactly one leaf (the genesis block's execution commitment), so $S(1)$ has one peak and size 1.

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

$$\text{merged} = \text{H}_{\text{node}}(\text{DOMAIN\_NODE} \parallel \text{left\_peak} \parallel \text{right\_peak}) \tag{6}$$

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"):

$$\text{bag}_0 = p_0 \tag{7}$$

$$\text{bag}_i = \text{H}_{\text{bag}}(\text{DOMAIN\_BAG} \parallel \text{bag}_{i-1} \parallel p_i) \quad \text{for } i = 1, \ldots, k-1 \tag{8}$$

$$\text{elog\_root} = \text{H}_{\text{root}}(\text{DOMAIN\_ROOT} \parallel \text{bag}_{k-1} \parallel \text{encode}_{64}(\text{elog\_size})) \tag{9}$$

Domain Separation Constants: To prevent hash collisions across different contexts, we define fixed domain prefixes:

- DOMAIN_LEAF = ''KASPA_MMR_LEAF_V1''

- DOMAIN_NODE = ''KASPA_MMR_NODE_V1''

- DOMAIN_BAG = ''KASPA_MMR_BAG_V1''

- DOMAIN_ROOT = ''KASPA_MMR_ROOT_V1''

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 left to right (from lowest position to highest position) and hashed in that sequence.

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** $i \leftarrow 1$ 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** $i \leftarrow 1$ to new_peaks.length -1 **do**
13:         bag ← H_{bag}(bag $\|$ new_peaks[i])
14:     **end for**
15:     elog_root ← H_{root}(bag $\|$ encode$_{64}$(new_size))
16:     **return** (MmrState{peaks: new_peaks, size: new_size}, elog_root)
17: **end function**

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.

3.5 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.

3.6 Integration with Consensus Execution

A critical distinction of our approach is that the MMR commitment is **derived from and validated alongside consensus execution**, not as a post-processing step. This section clarifies the integration points and their implications.

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_exec_root $\leftarrow$ MerkleRoot(accepted_tx_ids of B)
3: **if** expected_exec_root $\neq B$.exec_root **then**
4:      **return** BADEXECROOT
5: **end if**
6: **Step 2: MMR Leaf Computation**
7: $\text{leaf}_B \leftarrow H_{\text{leaf}}(\text{DOMAIN\_LEAF} \parallel P.\text{hash} \parallel B.\text{exec\_root})$
8: **Step 3: MMR Append Computation**
9: old_size $\leftarrow P$.elog_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 $\leftarrow H_{\text{node}}(\text{left\_peak} \parallel \text{current})$
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 $\leftarrow H_{\text{bag}}(\text{bag} \parallel \text{peaks}[i])$
23: **end for**
24: expected_elog_root $\leftarrow H_{\text{root}}(\text{bag} \parallel \text{encode}_{64}(\text{new\_size}))$
25: **Step 5: Verification**
26: **if** expected_elog_root $\neq B$.elog_root **then**
27:      **return** BADMMRROOT
28: **end if**
29: **if** new_size $\neq B$.elog_size **then**
30:      **return** BADMMRSIZE
31: **end if**
32: **return** VALID

3.6.1 Why Auxiliary MMR is Insufficient for L2

Previous approaches to historical commitments treat them as "auxiliary" structures—computed after consensus execution and validated separately. While this is simpler to implement, it has a fundamental limitation for L2 systems:

> *An auxiliary commitment can diverge from consensus execution without causing block rejection.*

Consider a hypothetical attack: a miner produces a block where the auxiliary MMR correctly chains to the parent, but the actual transactions executed differ from what the MMR commits to. If the auxiliary MMR is not validated against execution, this discrepancy goes undetected at the consensus level.

Our design prevents this by making `exec_root` and `elog_root` **consensus-validated fields**:

- `exec_root` is validated against the output of `calculate_utxo_state()`

- `elog_root` is validated against the MMR append operation using the validated `exec_root`

- A mismatch in either field causes **block rejection**—same as invalid PoW or UTXO commitment

3.6.2 Execution Pipeline Integration

The following pseudocode illustrates how MMR commitment integrates with the existing GHOSTDAG validation pipeline:

```
1   def validate_block(block, parent_state):
2       # Step 1: Standard UTXO validation (existing GHOSTDAG logic)
3       ctx = calculate_utxo_state(block, parent_state)
4
5       # Step 2: Validate UTXO commitment (existing)
6       if block.header.utxo_commitment != compute_utxo_commitment(ctx):
7           return REJECT_BAD_UTXO_COMMITMENT
8
9       # Step 3: Validate exec_root (NEW - execution order commitment)
10      expected_exec_root = merkle_root(ctx.accepted_tx_ids)
11      if block.header.exec_root != expected_exec_root:
12          return REJECT_BAD_ORDER_ROOT
13
14      # Step 4: Validate MMR commitment (NEW - history commitment)
15      parent_mmr = get_mmr_state(block.selected_parent)
16      leaf = H_leaf(DOMAIN_LEAF || block.selected_parent.hash || block.header.
            exec_root)
17      new_mmr, expected_elog_root = parent_mmr.append(leaf)
18
19      if block.header.elog_root != expected_elog_root:
20          return REJECT_BAD_MMR_ROOT
21      if block.header.elog_size != new_mmr.size:
22          return REJECT_BAD_MMR_SIZE
23
24      # All validations passed
25      return ACCEPT
```

Listing 4: Consensus Execution with MMR Validation

The key insight is that Steps 3 and 4 use the `ctx.accepted_tx_ids` from Step 1—the same deterministic sequence used for UTXO validation. This ensures the MMR commits to *actual execution*, not declared contents.

3.6.3 Atomicity with Chain State

The MMR state must be committed atomically with other chain state (UTXO set, acceptance data). Our implementation stores MMR state in the same database write batch:

```python
def commit_block(block, ctx, new_mmr_state):
    batch = WriteBatch()

    # All writes go in the same atomic batch
    batch.write(utxo_store, ctx.utxo_diff)
    batch.write(acceptance_store, ctx.acceptance_data)
    batch.write(mmr_state_store, block.hash -> new_mmr_state)
    batch.write(mmr_node_store, new_mmr_nodes)  # For proof generation

    database.atomic_write(batch)
```

Listing 5: Atomic State Commitment

This ensures that if a crash occurs mid-write, either all state is committed or none is—preventing inconsistency between UTXO state and MMR state.

3.6.4 Implications for L2 Systems

This tight integration provides strong guarantees for L2 systems:

1. **Execution Fidelity:** The `exec_root` commits to the *actual* execution order. An L2 replaying this order will reproduce the same state.

2. **Consensus-Level Security:** A wrong `exec_root` is as severe as a wrong UTXO commitment—the block is invalid. This is stronger than "the MMR might be wrong but the block is still valid."

3. **Proof Semantics:** When a proof verifies against `elog_root`, it proves not just "this transaction was in a block" but "this transaction was executed at this position in the global execution order."

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. 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.

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

To handle this:

- We maintain the MMR state per block (peaks and size) in our store
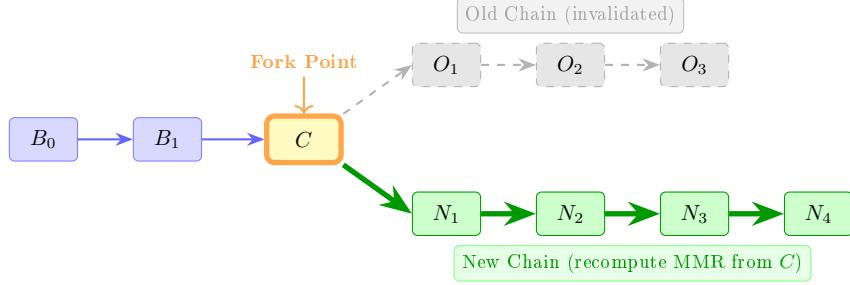
Figure 5: Chain reorganization handling. MMR state is retrieved from common ancestor C, and new chain blocks have their MMR states recomputed forward.

- 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.

4.2 Reorg Window and Checkpoint Policy

We parameterize the protocol's treatment of reorganization and finality:

Definition 4.1 (Protocol Parameters). • K_{reorg}: **Reorg Bound** — *maximum expected reorganization depth (blocks). Beyond this, proofs are considered "practically final." Typical value: 3,600 blocks ($\sim$6 minutes at 10 BPS).*

- $T_{checkpoint}$: **Checkpoint Period** — *interval between rolling checkpoints. Typical value: 86,400 blocks ($\sim$24 hours at 10 BPS).*

- $K_{finality}$: **Finality Depth** — *depth at which a block is considered irreversible for practical purposes. Often aligned with K_{reorg}.*

Proof Validity Windows:

1. **Within $K_{\mathbf{reorg}}$:** Proofs may become invalid due to reorganization. L2 systems SHOULD wait K_{reorg} confirmations before treating a proof as final, or be prepared to re-request updated proofs.

2. **Beyond $K_{\mathbf{reorg}}$:** Proofs are "practically final"—reorganization of this depth would require attacking the consensus itself.

3. **Beyond $T_{\mathbf{checkpoint}}$:** If rolling is used, nodes may need pre-generated proofs or archive node services.

Checkpoint Alignment: For optimal interoperability, checkpoints SHOULD align with protocol-level posterity headers (if any). This enables:

- Consistent proof composability across nodes

- Known anchor points for historical verification

- Simplified disaster recovery procedures

Table 4: Recommended Protocol Parameters (10 BPS)

Parameter	Value	Time	Rationale
K_{reorg}	3,600	$\sim$6 min	Conservative reorg bound
K_{finality}	3,600	$\sim$6 min	Aligned with reorg bound
$T_{\text{checkpoint}}$	86,400	$\sim$24 hr	Aligned with pruning period

4.3 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 ($\sim$1 year at 10 BPS): $\sim$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 ($\sim$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 $\sim$20 peaks each $\approx$ 55 MB

4.4 Pruning and Checkpointing

GHOSTDAG 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 `elog_root`.

4.5 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.

Critical Invariant: Rolling is a *storage optimization only*—it does not alter consensus commitments. The `elog_root` in block headers is always computed from the full MMR (conceptually from genesis), regardless of what a particular node stores locally.

> *"Rolling does not alter consensus commitments; it only affects the availability of historical witnesses."*

4.5.1 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)

Consensus Remains Unaffected: A node that has rolled its MMR can still:

- Validate new blocks (only requires parent's MMR state, which is recent)

- Compute `elog_root` correctly (same algorithm, same result)

- Participate fully in consensus

What the node *cannot* do after rolling is generate proofs for blocks before the checkpoint. This is a data availability limitation, not a consensus limitation.

4.5.2 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

Proof Composability: A key property of MMR proofs is that they can be *spliced* across checkpoints. Given:

- Proof π_1: transaction $\rightarrow$ `elog_root` at checkpoint C_1

- Proof π_2: checkpoint C_1's state $\rightarrow$ `elog_root` at checkpoint C_2

A verifier holding `elog_root` at C_2 can verify the transaction by composing π_1 and π_2. This enables a hierarchical proof ecosystem where archive nodes store proof segments and users compose them on demand.

4.5.3 Node Capability Tiers

Rolling creates a natural tiering of node capabilities:

Node Type	Consensus Participation	Proof Generation
Pruned Node	Full	None (no MMR nodes stored)
Rolling Node	Full	Recent blocks only
Archive Node	Full	All historical blocks

All node types participate equally in consensus. The difference is purely in their ability to *serve proofs* to external parties.

4.5.4 Storage Trade-offs

Strategy	Storage (per year)	Proof Availability
Full MMR (no rolling)	$\sim$20 GB/year cumulative	All historical proofs
Daily checkpoints	$\sim$500 MB/year	Recent + checkpoint-based
Weekly checkpoints	$\sim$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 GHOSTDAG implementations already handle block body pruning—the MMR commitment in headers enables verification even when underlying data is pruned, as long as proofs were generated before pruning.

4.5.5 Proof Availability: Formal Model

We now formalize the proof availability guarantees, distinguishing between **protocol guarantees** and **economic/social assumptions**.

Definition 4.2 (Verification vs. Generation). • **Proof Verification:** *Given a proof π and trusted* `elog_root`, *determine if π is valid. Requires only the proof data and header.*

- **Proof Generation:** *Given a transaction tx and block B, produce a valid proof π. Requires access to MMR internal nodes.*

Key Property: *Verification is always possible given valid proof data; generation depends on data availability.*

Definition 4.3 (Availability Models). *We define two availability models:*
Model A: Bounded-History Verification (Protocol Guarantee)

- *For blocks within the last $T_{checkpoint}$ blocks ($\sim$24 hours), all full nodes* CAN *generate proofs*

- *This is a **protocol guarantee**: the protocol specifies minimum retention*

- *Light clients can obtain proofs for recent transactions from any full node*

Model B: Historical Verification (Economic Assumption)

24

- *For blocks older than $T_{checkpoint}$, proof generation requires archive nodes*

- *This is an **economic assumption**: at least one honest archive node is online and reachable*

- *The protocol does not guarantee proof availability for historical blocks*

Table 5: Proof Availability Guarantees

Block Age	Proof Genera-tion	Guarantee Type	Failure Mode
$< T_{\text{checkpoint}}$	Any full node	Protocol	None (guaranteed)
$\geq T_{\text{checkpoint}}$	Archive nodes only	Economic	No archive online $\Rightarrow$ no new proofs
Any age	—	—	Pre-generated proofs remain valid forever

4.5.6 Node Capability Tiers

Definition 4.4 (Node Capability Classes). *1. **Archive Node:** Stores complete MMR from genesis. Can generate proofs for any block. Storage: $\sim$20 GB/year.*

*2. **Full Node (Rolling):** Stores MMR state from last checkpoint. Can generate proofs for recent blocks, verify any proof. Storage: bounded.*

*3. **Light Node:** Stores only headers. Can verify proofs, cannot generate them.*

4.5.7 Minimum Data Retention Requirements

- MUST retain: All block headers (including `exec_root`, `elog_root`, `elog_size`)—required for consensus

- MUST retain: MMR peaks at each checkpoint (for proof composability across checkpoints)

- SHOULD retain: MMR internal nodes for last $T_{\text{checkpoint}}$ blocks (for proof generation)

- MAY prune: MMR internal nodes for blocks before checkpoint

- MAY prune: Block bodies per existing pruning rules

4.5.8 Failure Modes and Mitigations

Failure Mode 1: No archive node available

- **Impact:** Cannot generate new proofs for blocks older than $T_{\text{checkpoint}}$

- **Mitigation:** Pre-generate and cache proofs for important transactions before pruning

- **Recovery:** If *any* archive comes online later, historical proofs become available again

Failure Mode 2: Malicious archive provides invalid proof

- **Impact:** None—light client can detect and reject invalid proofs via verification

- **Property:** Archives cannot forge proofs; they can only refuse to serve

Failure Mode 3: Light client cannot find any proof source

- **Impact:** Proof retrieval fails (data availability issue, not security issue)

- **Mitigation:** Multiple independent archives, incentivized proof services

4.5.9 Economic Incentives (Out of Scope)

We note that a complete system may include economic incentives for archive operation:

- Proof generation fees paid to archive operators

- Staking/slashing for archive availability guarantees

- Protocol-level archival rewards

These mechanisms are **outside the scope** of this paper. Our contribution is the cryptographic scheme; incentive design is left to implementation.

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 Two-Layer Proof Model

To prove that a transaction tx was **executed at a specific position** in the global execution order, we provide a two-layer proof structure:

1. **Intra-Block Execution Proof:** Proves tx was executed at position i within block B's accepted transaction list ($0 \leq i < |\text{AcceptedTxList}(B)|$)

2. **Inter-Block MMR Inclusion Proof:** Proves block B's execution commitment is at leaf index j in the execution log MMR ($j =$ block's position in selected chain)

5.1.1 What Exactly Is Proven?

We are precise about what the proof establishes:

Proposition 5.1 (Execution Position Proof)**.** *A valid two-layer proof establishes the following statement:*

> *"Transaction tx is the i-th executed transaction in block B, where block B is the j-th block in the selected parent chain (with $j = leaf\_index$)."*

*This gives a **hierarchical position** (j, i): block index j, intra-block index i.*

Important Clarification: The proof does *not* directly establish a "global transaction index" (e.g., "the 1,234,567-th transaction ever executed"). Computing such an index would require knowing the cumulative transaction count up to block $j - 1$, which is not committed in our current scheme.
Options for Global Indexing (if needed):

1. **Prefix-Sum Commitment:** Add a `cum_tx_count` field to each block header, committing the total transactions executed up to (and including) that block. Then global index = $\text{cum\_tx\_count}(j - 1) + i$.

2. **Encode Count in Leaf:** Extend the MMR leaf to include block's transaction count: $L(B) = H_{\text{leaf}}(\text{parent\_hash}\|\text{exec\_root}\|\|\text{AcceptedTxList}|)$. A verifier can sum counts along the proof path.

3. **Accept Hierarchical Position:** For many L2 use cases, (j, i) is sufficient—L2 contracts can maintain their own mapping if needed.

For this paper, we adopt option (3): the hierarchical position (j, i) is the canonical proof output. This keeps the scheme minimal while remaining sufficient for most applications (transaction ordering, dispute resolution, bridge verification).

This two-layer structure is the **minimal proof** that an L2 system needs to verify execution position. It is more powerful than skip-list chain membership proofs (which only prove block inclusion) and more efficient than linear chain traversal (which requires $\mathcal{O}(n)$ operations).
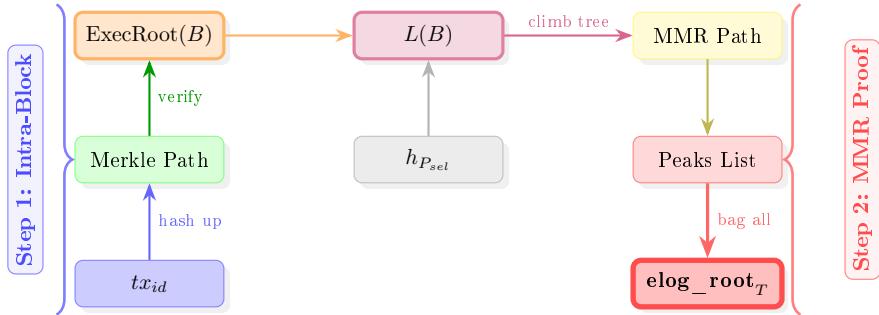

Figure 6: Two-step proof verification process. Step 1 verifies the transaction is in block B via Merkle proof. Step 2 verifies block B is in the MMR via MMR proof.

We formalize a proof as a tuple $(txId, B, \mathrm{IntraProof}, \mathrm{MMRProof}, T)$ 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 ExecRoot 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 (`exec_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 6: Intra-Block Proof Structure

MMR Inclusion Proof Format:

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

Listing 7: MMR Proof Structure

Proof Size Optimization: Peak Compression

The `peaks` field contains all MMR peak hashes. For a chain with n blocks, there are at most $\lceil \log_2 n \rceil$ peaks. At 10 BPS over 10 years ($\sim$3 billion blocks), this is $\sim$32 peaks $\times$ 32 bytes = 1 KB.

For further optimization, implementations MAY use one of:

1. **Full Peaks (Default):** Include all peak hashes. Simple, slightly larger proofs.

2. **Bagged Peaks Root:** The verifier already knows `elog_root`, which is derived from peaks. The proof includes only peaks *not* on the verification path. The verifier reconstructs missing peaks during path computation.

3. **Compressed Peaks Commitment:** Store a Merkle root of peaks in the proof; include Merkle proof for needed peaks. Trades computation for bandwidth.

For this paper, we use option (1) for simplicity. The proof size difference is marginal ($\sim$500 bytes), and full peaks simplify verification logic.

5.2 Proof Size and Efficiency

Theorem 5.2 (Proof Size Bound). *For a transaction in block B with m transactions, and MMR with N leaves, the total proof size is:*

$$ProofSize = \mathcal{O}(\log m + \log N) \times 32 \; bytes \tag{10}$$

Table 6: Proof Size Estimates for Various Chain Lengths at 10 BPS

Chain Length	Time	MMR Depth	Est. Proof Size	Verification Hashes
10^6 blocks	$\sim$1 day	20	$\sim$800 bytes	$\sim$25
10^7 blocks	$\sim$12 days	24	$\sim$900 bytes	$\sim$30
10^8 blocks	$\sim$4 months	27	$\sim$1.0 KB	$\sim$35
3×10^8 blocks	$\sim$1 year	28	$\sim$1.1 KB	$\sim$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 7: Proof Size (Hash Operations) vs Age of Transaction

Blocks Ago (d)	Linear Hash Ops	MMR Hash Ops	Improvement
100	$\sim$100	$\sim$7–8	93% fewer
1,000	$\sim$1,000	$\sim$10–11	99% fewer
10,000	$\sim$10,000	$\sim$13–14	99.9% fewer
100,000	$\sim$100,000	$\sim$17–18	99.98% fewer
1,000,000	$\sim$1,000,000	$\sim$20–21	99.998% fewer

Even at moderate distances like 10,000 blocks ($\sim$1,000 seconds at 10 BPS), the difference is stark: 10,000 vs $\sim$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 $\sim$30 hashes is feasible. Each hash costs approximately $\sim$1920 gas (for Keccak-256: $\sim$ 8 gas/byte $\times$ 64 bytes). Therefore:

- MMR proof (30 hashes): $\sim$57,600 gas—very manageable (typical ETH transfer costs $\sim$21,000 gas)

- Linear 100k hash proof: $\sim$192 million gas—exceeds block gas limit ($\sim$30M) by 6$\times$

Algorithm 3 Complete Proof Verification

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

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:

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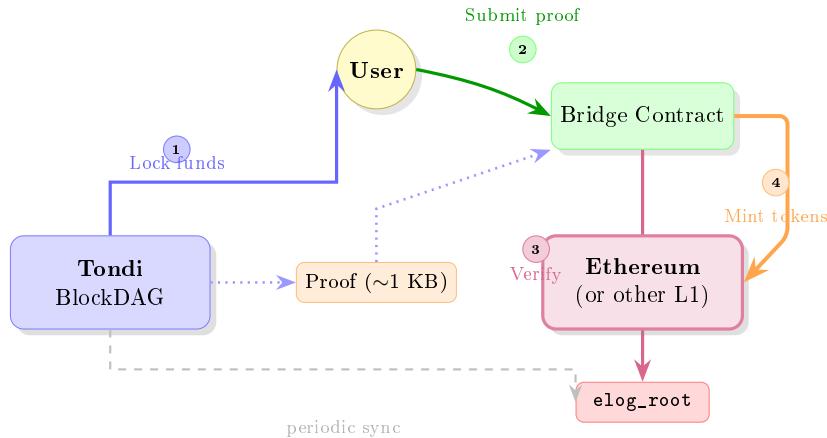
Figure 7: Trustless bridge architecture using MMR proofs. The bridge contract stores only the latest `elog_root` and verifies inclusion proofs on-chain with ~30 hash operations.

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 `elog_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:

At 100,000 blocks depth (~2.8 hours at 10 BPS), linear verification 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 a GHOSTDAG-based blockchain ecosystem, having cryptographic receipts means:

- **Simple Payment Verification (SPV):** A client only needs to track the chain of `elog_root` in headers and for any transaction of interest, get a proof from a peer and verify it

Table 8: On-Chain Verification Cost Comparison (Ethereum)

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

- **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 `elog_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 `exec_root` against its header and then verifies that block's inclusion in the MMR. If any block's data was tampered with, its `exec_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 `elog_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 `elog_root`. Then download actual transactions for say 1 out of each 100 blocks (randomly chosen), verify those blocks' `exec_root`s 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 `elog_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 `elog_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 `elog_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 `exec_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 9: 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
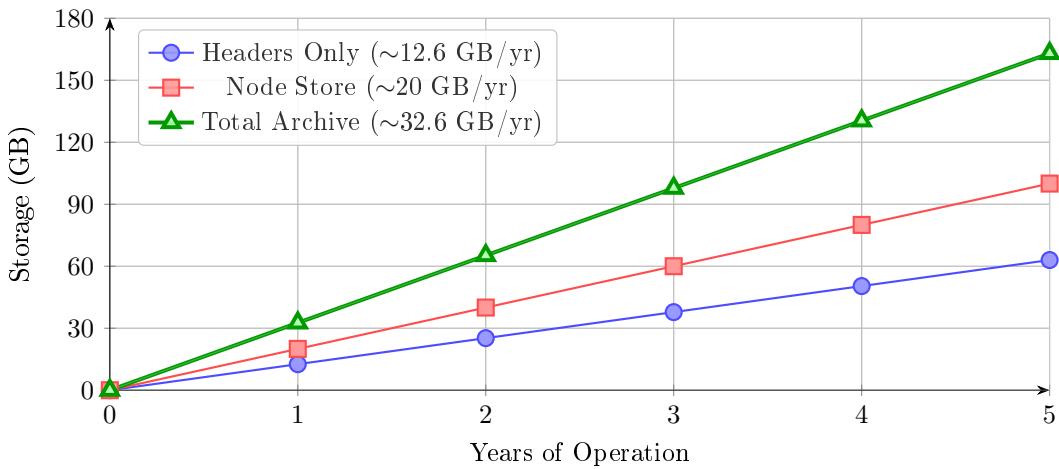
Figure 8: Storage growth over time at 10 BPS. Headers grow at ~12.6 GB/year; optional node store for proof generation adds ~20 GB/year. Even after 5 years, total storage remains under 165 GB—manageable for modern hardware.

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, a GHOSTDAG-based blockchain gains a built-in universal proof system for its transaction history.

8 Related Work

We situate our contribution within the broader landscape of blockchain proof systems and history commitments.

8.1 Merkle Mountain Ranges in Blockchain Systems

Merkle Mountain Ranges were introduced by Todd [7] as an append-only accumulator structure. MMRs have been adopted in several blockchain systems:

- **Grin/MimbleWimble** [6]: Uses MMR for UTXO set commitment, enabling efficient pruning while maintaining cryptographic proofs of historical UTXOs.

- **Nervos CKB** [5]: Employs MMR for transaction commitment, providing $\mathcal{O}(\log n)$ inclusion proofs.

- **Herodotus Protocol** [3]: Uses MMR as a historical block hash accumulator for cross-chain verification, demonstrating MMR's applicability to bridge protocols.

Our contribution differs in that we use MMR to commit to *execution order* (the sequence of accepted transactions), not just block or UTXO existence. This requires careful integration with GHOSTDAG's ordering semantics.

8.2 Logarithmic Proof Protocols

Several works address logarithmic-complexity blockchain proofs:

- **FlyClient** [11]: Provides super-light clients for PoW blockchains using probabilistic sampling and Merkle Mountain Ranges. FlyClient proves chain validity with $\mathcal{O}(\log n)$ headers. Our work is complementary: FlyClient proves chain membership; we prove execution order within that chain.

- **NiPoPoWs** [12]: Non-Interactive Proofs of Proof-of-Work use superblock interlinking to achieve succinct proofs. Like FlyClient, NiPoPoWs focus on chain connectivity, not transaction ordering.

- **Utreexo** [15]: Provides compact UTXO set commitments, reducing node storage requirements. Utreexo addresses state commitment; we address history commitment—orthogonal concerns.

8.3 DAG Consensus and Linearization

Our work builds on GHOSTDAG [4], which provides a specific linearization rule for blockDAGs:

- **PHANTOM/GHOSTDAG** [4]: Defines blue set selection and mergeset ordering, producing a deterministic total order of blocks. Our scheme commits to the *transaction-level* order derived from this block order.

- **SPECTRE** [16]: Provides fast confirmation but only partial ordering (pairwise). Not directly applicable to our scheme, which requires total ordering.

- **Conflux** [17]: Uses Tree-Graph structure with pivot chain selection. Our approach could adapt to Conflux by binding to its pivot chain and block ordering.

8.4 Transaction Ordering Commitments

Ordering commitments are crucial for Layer-2 systems:

- **Ethereum's Transaction Receipts**: Each block commits to a transaction receipt trie, providing ordering within blocks but not global ordering.

- **Rollup Sequencing**: Optimistic and ZK rollups rely on L1 ordering for dispute resolution. Our scheme provides the ordering proofs these systems need.

- **MEV and Fair Ordering**: Recent work on MEV mitigation (e.g., Flashbots, threshold encryption) addresses *preventing* ordering manipulation; we address *proving* the ordering that occurred.

8.5 Positioning of Our Contribution

Table 10: Related Work Comparison

System	Commits To	Proof Type	Complexity	DAG Support
FlyClient	Chain validity	Probabilistic	$\mathcal{O}(\log n)$	No
NiPoPoWs	Chain connectivity	Deterministic	$\mathcal{O}(\log n)$	No
Utreexo	UTXO set	Deterministic	$\mathcal{O}(\log n)$	N/A
Grin MMR	UTXO existence	Deterministic	$\mathcal{O}(\log n)$	No
Ours	**Execution order**	**Deterministic**	$\mathcal{O}(\log n)$	**Yes**

Our unique contribution is combining MMR-based logarithmic proofs with GHOSTDAG's deterministic execution ordering, producing a **consensus-level execution log** suitable for L2 and cross-chain applications.

9 Comparison with Alternative Approaches

In this section, we provide a detailed comparison between our MMR-based unified approach and two alternative approaches commonly considered for blockDAG protocols: linear sequencing commitments (chain-based ordering) and skip-list based proofs of chain membership (PoChM). Understanding the design philosophy, strengths, and limitations of each approach illuminates why our unified solution represents an optimal path forward.

9.1 Linear Sequencing Commitment

9.1.1 Design Philosophy and Mechanism

Linear sequencing commitment addresses a fundamental need for Layer-2 networks: the ability to trustlessly verify the *order* of transaction acceptance on L1. The key insight is that L2 networks rely on the base layer 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 header field called `SequencingCommitment`, calculated as:

$$SC_B = H(SC_{P_{sel}(B)} \parallel \text{AcceptedIDMerkleRoot}_B) \tag{11}$$

where $P_{sel}(B)$ is the selected parent of block B.
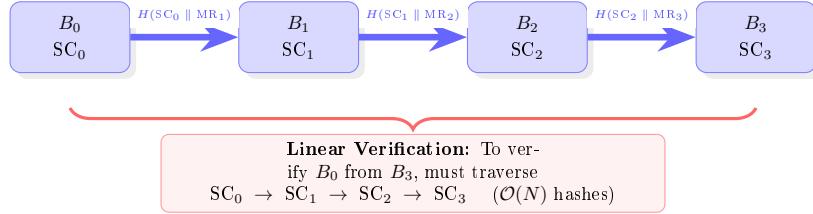
Figure 9: Linear Sequencing Commitment. Each block's SC depends on its parent's SC, creating a sequential chain. Verification requires traversing all intermediate commitments.

Additionally, this approach proposes that the `AcceptedIDMerkleRoot` should preserve *canonical order* rather than sorting transactions by hash. Hash-sorting was originally an optimization for exclusion proofs, but in practice, this feature is rarely used.

9.1.2 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)

9.1.3 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. This 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. **Requires Separate Proof Mechanism:** Linear sequencing alone does not provide efficient proofs of chain membership; it assumes full header availability.

9.2 Skip-List Based Proof of Chain Membership (PoChM)

9.2.1 Design Philosophy and Mechanism

Skip-list based PoChM addresses a complementary problem: providing cryptographically verifiable proofs that a transaction was included in the blockchain, especially after pruning has occurred. The key insight is 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."

This approach introduces the concept of **Posterity Headers**—selected chain block headers sampled at regular intervals (e.g., every 24 hours) and stored indefinitely. The core mechanism adds a **PoChM Merkle Root (PMR)** to each header:

Definition 9.1 (PMR Construction). *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 $(2^0, 2^1, 2^2, \ldots)$.
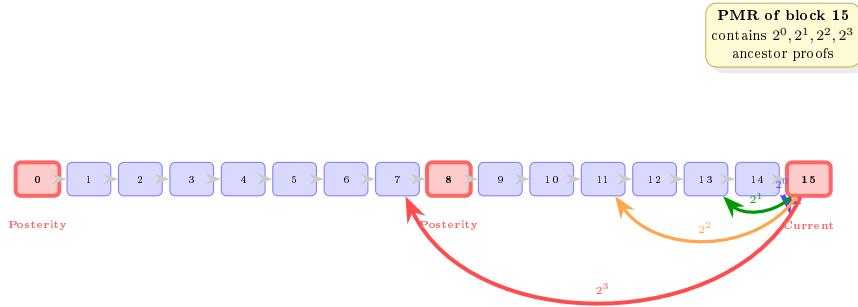

Figure 10: Skip-list PoChM Structure with exponential skip pointers (2^i). Each block's PMR contains Merkle proofs of ancestors at distances $2^0, 2^1, 2^2, \ldots$ The proof path uses binary decomposition of the distance.

9.2.2 Proof Construction Algorithm

The PoChM generation algorithm uses binary decomposition:

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

2. While $d > 0$:
   - Let $i = \lfloor \log_2(d) \rfloor$
   - Append $PMR(C, i)$ to proof
   - $d \leftarrow d - 2^i$; $C \leftarrow parent(C, 2^i)$

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

9.2.3 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

36

9.2.4 Limitations

1. **Does Not Commit Transaction Order:** Skip-list PoChM provides proofs that a block was in the selected chain, but not the canonical ordering of transactions. The proof is about chain membership, not transaction sequence.

2. **Requires Separate Structure for Ordering:** To prove both chain membership *and* transaction order, PoChM must be combined with a linear sequencing commitment, resulting in two parallel commitment structures.

3. **Posterity Header Dependency:** The scheme depends on posterity headers (e.g., 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.

9.3 Comparative Analysis: Deep Insights

9.3.1 The Fundamental Tension

Linear sequencing and skip-list PoChM represent two orthogonal responses to the same underlying challenge: how to provide verifiable historical information in a pruning blockchain.

- **Linear sequencing** prioritizes *ordering integrity*—ensuring the canonical sequence of transactions is cryptographically committed and verifiable.

- **Skip-list PoChM** 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 *and* proof of its position in the global order.

9.3.2 The Architectural Incompatibility

Implementing linear sequencing and skip-list PoChM 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 proving block chain membership
   - A Merkle proof for transaction within block's accepted ID root
   - A chain of SequencingCommitments if ordering matters

   The third component reintroduces linear complexity, negating PoChM's benefits for order-sensitive applications.

3. **Conceptual Overlap Without Synergy:** Both mechanisms touch block headers and validation rules, but don't reinforce each other's proofs.

9.3.3 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 ExecRoot.

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 `exec_root` commits to transaction ordering, while the `elog_root` enables efficient proofs. These naturally compose rather than compete.

9.3.4 The Fundamental Philosophical Difference

Beyond technical metrics, there is a deeper distinction:

> *Traditional approaches treat MMR as a "compressed representation of history."*
>
> *Our approach treats MMR as a "canonical interface to consensus execution results."*

This is not merely an implementation detail—it represents different answers to the question: *What is the MMR's role in the protocol?*

- **Auxiliary Log View:** The MMR is computed *after* consensus decides block validity. It is a convenience structure for external consumers. Consensus could, in principle, work without it.

- **Consensus Output View (Our Approach):** The `exec_root` and `elog_root` are *outputs of consensus execution*, validated alongside UTXO commitments. A block with incorrect MMR fields is invalid at the consensus level.

The practical implication: in our design, an L2 system can treat the `elog_root` as *the canonical interface* through which it queries the blockchain's execution history. It is not merely "a way to prove things"—it is *the* authoritative commitment to what was executed.

9.3.5 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 9.2 (Gas Cost Bounds). *For a proof of depth D blocks:*

$$Gas_{Linear} = 1920 \times D \tag{12}$$

$$Gas_{PoChM} \approx 1920 \times (2\log_2 D \times \lceil \log_2 \log_2 D \rceil + 1) \tag{13}$$

$$Gas_{MMR} \approx 1920 \times (\log_2 D + k) \tag{14}$$

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):

- Linear: 192 million gas (exceeds block limit by 6$\times$)

- Skip-list PoChM: $\sim$104,000 gas (feasible)

- Execution-MMR: $\sim$61,000 gas (feasible, 41% lower than PoChM)

Table 11: Detailed Feature Comparison: Linear Sequencing, Skip-List PoChM, and MMR-Based Approach

| Aspect	Linear Seq.	Skip-List PoChM	Execution-MMR (Ours)
Core Functionality			
Commits TX Order	✓ (chain hash)	✗	✓ (via ExecRoot)
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)
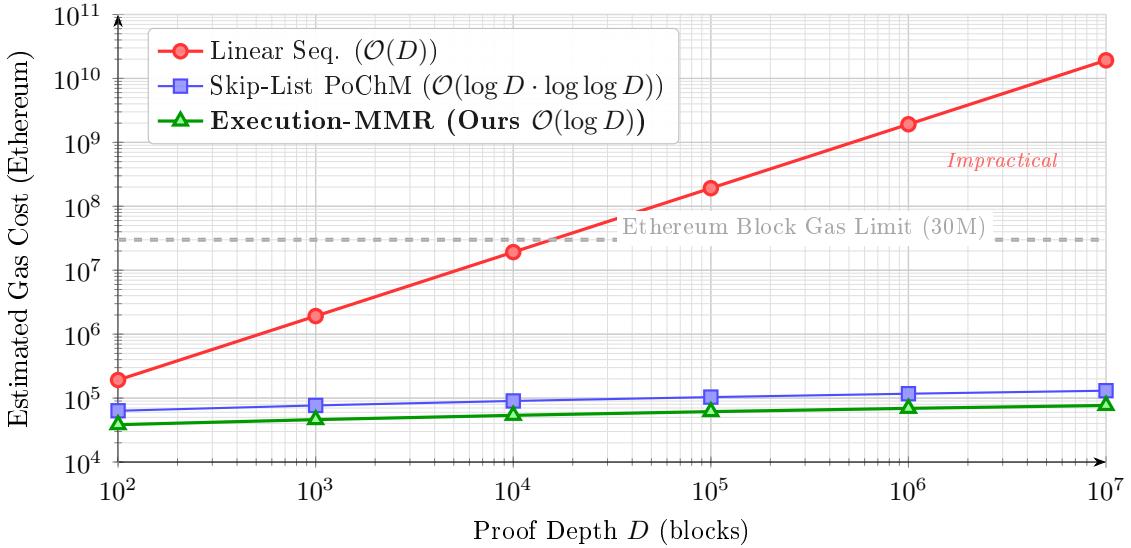
Figure 11: On-chain verification gas cost comparison. Linear sequencing's growth makes it impractical beyond ~15,000 blocks (crosses gas limit). Both skip-list PoChM and MMR remain feasible at any depth, with MMR offering ~40% lower costs due to simpler proof structure.

9.3.6 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 and hash chains.

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 skip-list PoChM, 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.

9.4 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.

> *Linear sequencing asks: "How do we commit to the order?"*
> *Skip-list PoChM asks: "How do we prove membership efficiently?"*
> *Execution-MMR answers both: "By embedding the order in an efficient accumulator."*

Implementing linear sequencing and skip-list PoChM 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.

The "Consensus Impact" comparison (Table 13) highlights our key architectural contribution: making the MMR a *first-class consensus artifact* rather than an auxiliary structure. This distinction has profound implications:

1. **Security:** A consensus-level MMR commitment cannot be "wrong but tolerated"—it causes block rejection

2. **Trust Model:** L2 systems can rely on MMR proofs with the same trust as UTXO commitments

3. **Unified Design:** We don't implement linear sequencing alongside MMR—the `exec_root` *is* the sequencing commitment

Table 12: Summary Comparison of Transaction Commitment Approaches

Capability	Linear Seq.	Skip-List PoChM	Execution-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

Table 13: Consensus Impact Comparison: The Key Distinction

Dimension	Linear Seq.	Skip-List (aux)	Execution-MMR (Ours)
Modifies header	✓	✓	✓
Affects consensus execution	✗	✗	✓
Replaces linear sequencing	—	✗	✓
MMR is first-class artifact	✗	✗	✓
Proof is protocol-level defined	✗	Partial	✓
Rolling is consensus-compatible	N/A	Ambiguous	✓ (explicit)

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. The adoption of MMR commitments in GHOSTDAG-based protocols would be among the first in live L1 blockchains, and we believe it will greatly enhance ecosystem capabilities.

10 Implementation and Deployment

10.1 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.

10.2 Miner Integration

Miners need to upgrade their block template generation:

1. Compute `exec_root` from accepted transactions

2. Fetch current MMR state from node (peaks and size)

3. Compute new leaf hash and updated `elog_root` and `elog_size`

4. Populate these in the header, then perform PoW

10.3 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

10.4 Security Analysis and Threat Model

We analyze the security of the execution log scheme under the standard cryptographic assumptions.

10.4.1 Threat Model

Attacker Goals:

1. **History Forgery:** Convince a verifier that a transaction was executed when it was not, or at a different position

2. **History Omission:** Convince a verifier that a transaction was *not* executed when it actually was

3. **Order Manipulation:** Convince a verifier of a different execution order than what actually occurred

   Attacker Capabilities:

- Can construct arbitrary proofs and data structures

- Cannot break cryptographic hash functions (collision resistance assumed)

- Cannot rewrite confirmed blockchain headers (would require consensus attack)

10.4.2 Security Properties

Proposition 10.1 (History Forgery Resistance). *An attacker cannot produce a valid proof for a transaction that was not executed without either:*

1. *Finding a hash collision in the Merkle tree or MMR construction, or*

2. *Producing a block header with a forged* `elog_root` *that is accepted by consensus (requires mining attack)*

   Proof Sketch: The two-layer proof binds transaction → `exec_root` → MMR leaf → `elog_root` → block header. Each binding is via cryptographic hashing. Forging any component requires either a hash collision or consensus-level attack.

Corollary 10.2 (Consensus-Equivalent Security). *The security of execution log proofs is equivalent to consensus security. Specifically:*

> Forging a historical execution proof requires the same computational effort as forging a block header—i.e., attacking consensus itself.

This is the key insight: the execution log is not an "auxiliary data structure that could be wrong." It is a consensus output with consensus-level security guarantees.

10.4.3 Specific Attack Vectors

- **Collision Resistance:** Domain separation (distinct prefixes for leaf, node, bag, root hashes) prevents cross-context collisions. Under the random oracle model, finding any collision requires $\mathcal{O}(2^{128})$ work for 256-bit hashes.

- **Replay Protection:** The `elog_root` is embedded in block headers. Tampering requires re-mining the block and all descendants—a consensus-level attack with cost proportional to chain security.

- **Denial of Service:** Proof verification requires $\mathcal{O}(\log n)$ hash operations (~ 30 hashes for a 10-year history). This is fast enough to reject invalid proofs without resource exhaustion.

- **Reorg Exploitation:** An attacker could attempt to produce two different histories with the same `elog_root`. This requires an MMR collision—finding two different leaf sequences that produce the same root—which reduces to hash collision finding.

10.4.4 Trust Assumptions

For a verifier trusting an execution proof:

1. **Block Header Authenticity:** The verifier must obtain authentic block headers (e.g., from trusted source, or by verifying PoW chain)

2. **Hash Function Security:** The hash function (e.g., BLAKE3, SHA-256) must be collision-resistant

3. **No Long-Range Reorg:** If the proof references a block at depth $> K_{\text{finality}}$, the verifier assumes no reorganization past that depth

These are the same assumptions required for trusting any blockchain data, making execution log proofs no weaker than consensus itself.

11 Conclusion

We have presented an *Execution-MMR Sequencing Commitment* scheme for GHOSTDAG-based blockDAGs that fundamentally redefines how external systems interact with consensus. The MMR is not an auxiliary history log, but a **first-class consensus artifact** that serves as the canonical interface to the blockchain's execution results.

11.1 Summary of Results

Our scheme adds a small overhead (40 bytes per block) to block headers, which is negligible relative to block sizes and bandwidth. Based on experiments with Tondi (a GHOSTDAG implementation running at 10 BPS), 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, a GHOSTDAG-based blockchain gains:

1. **Verifiable Execution Order:** Every block locks in the exact order of *executed* transactions (not just included transactions). The `exec_root` is derived from the same `accepted_tx_ids` that determines UTXO state, ensuring perfect alignment between commitment and execution.

2. **Compact Execution Proofs:** Any transaction's execution position can be proven with a proof on the order of 0.5–1 KB, regardless of how old it is. The two-layer proof structure (intra-block + inter-block) provides the *minimal* proof an L2 system needs.

3. **Consensus-Level Security:** Unlike auxiliary logs, a wrong `exec_root` or `elog_root` causes block rejection at the consensus level. L2 systems can trust these commitments with the same confidence as UTXO commitments.

4. **Unified Ordering and Proof Mechanism:** The `exec_root` achieves the ordering commitment goal, while the `elog_root` achieves the logarithmic proof goal. These are not separate mechanisms—they are a single, coherent design.

5. **Efficient Sync & Recovery:** Nodes can sync faster and recover from data loss using the MMR commitments to validate bulk data from untrusted sources. Rolling MMR is explicitly designed to be consensus-compatible, with clear semantics for proof availability.

6. **Self-Authenticating Execution History:** The whole history becomes verifiable against the latest block's `elog_root`. Any single block's wrong execution breaks the root instantly.

11.2   The Broader Implication

The fundamental insight of this work is that **ordering commitment and efficient proofs are not independent problems**—they are two facets of the same challenge: providing verifiable access to consensus execution results.

> *Previous approaches asked: "How do we prove things about the chain?"*
> *Our approach asks: "How do we make consensus execution externally queryable?"*

By treating the MMR as a consensus output rather than an auxiliary structure, we position GHOSTDAG-based blockchains to be the foundation for a new generation of L2 systems that can trustlessly verify L1 execution with minimal overhead. The `elog_root` becomes *the* authoritative interface through which external systems query what the blockchain executed—not merely a convenience for generating proofs.

The proof size remains around 1 KB even for millions of blocks, and on-chain verification requires only ∼30 hashes (∼60,000 gas on Ethereum). This makes practical what was previously impossible: trustless cross-chain verification of transactions years old, at the same cost as verifying yesterday's transactions.

11.3   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 GHOSTDAG-based protocols continue to push the limits of blockDAG performance, features like this ensure that their security and usability for advanced applications keep pace. We anticipate this will accelerate development of L2 ecosystems and serve as a model for other projects seeking efficient historical proof systems.

References

[1] Linear Sequencing Commitments for BlockDAG Transaction Ordering. *BlockDAG Research*, 2024.

[2] S. Wyborski, "Proof of Chain Membership: Logarithmic Transaction Inclusion Proofs," *BlockDAG Improvement Proposals*, 2023.

[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] Y. Sompolinsky, S. Wyborski, and A. Zohar, "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.

[15] T. Dryja, "Utreexo: A Dynamic Hash-Based Accumulator Optimized for the Bitcoin UTXO Set," *Cryptology ePrint Archive*, Report 2019/611, 2019.

[16] Y. Sompolinsky, Y. Lewenberg, and A. Zohar, "SPECTRE: A Fast and Scalable Cryptocurrency Protocol," *Cryptology ePrint Archive*, Report 2016/1159, 2016.

[17] C. Li, P. Li, D. Zhou, W. Xu, F. Long, and A. Yao, "Scaling Nakamoto Consensus to Thousands of Transactions per Second," *arXiv preprint arXiv:1805.03870*, 2018.