Polymorphic Merkle Logs: Achieving Seamless Algorithm Lifecycle Upgrades without Ledger Fragmentation

Timothy DeHerrera
Cyphrpunk LLC
timothy@cyphrpunk.com

Zachary Collier
Cyphrpunk LLC
zach@cyphrpunk.com

Abstract

Verifiable append-only logs rely on cryptographic hashes for historical immutability. However, executing a hash transition on a live log traditionally forces a trilemma: fracturing the ledger into incompatible forks, accepting weakest-link algorithmic dependencies through chimeric constructions, or incurring a prohibitive $\mathcal{O}(N)$ retroactive compute penalty. We introduce the *Polymorphic Merkle Log* (PML), a novel abstract class of cryptographically agile authenticated data structures that supports multiple projected cryptographic hash profiles over a single, unified tree topology. We present the *Epoch Merkle Log* (EML) as a concrete, RFC 9162-compliant instantiation of this paradigm, partitioning algorithm lifecycles into discrete, null-padded chronological epochs. The EML achieves strictly algorithm-independent verification—each hash algorithm's projection yields a complete, independently verifiable Merkle tree—while enabling $\mathcal{O}(\log N)$ hash transitions with zero retroactive computation. We formalize the PML algebra—proving that shift-reduce duality is a universal structural property of append-consistent tree layouts—and verify the correctness of the EML instantiation via a machine-checked model in Lean 4 (15 definitions, 5 theorems) that decouples structural tree topology from cryptographic digests via a unique algebra homomorphism. We provide a production Rust implementation validated by property-based testing, fault injection, and complexity regression, demonstrating $\mathcal{O}(1)$ amortized appends, $\mathcal{O}(\log N)$ proofs, and zero-overhead bandwidth compression via deterministic proof elision.

1 Introduction

Long-lived authenticated data infrastructures are structurally bound to the lifecycles of their underlying cryptographic hash functions. Whether driven by hash decay, regulatory compliance mandates, or post-quantum migrations, cryptographic agility is essential for ledger longevity. Traditionally, executing an algorithm transition on a live log forces an unviable trilemma:

1. **Fork the ledger.** Abandon the old log and start a new one. This preserves history but fragments the timeline: verifiers must track multiple disjoint log identities, and the append-only invariant holds only within each shard. Certificate Transparency [10] and Sigsum [18] operate in this mode.

2. **Accept weakest-link security.** Combine multiple algorithms via runtime negotiation or layered signatures. The security of the composite log degrades to its weakest constituent, creating vulnerability surfaces for downgrade attacks (e.g., in TLS or JWT) and algorithm confusion [1, 11].

3. **Pay $\mathcal{O}(N)$ retroactive compute.** Re-hash the entire historical log under the new primitive. In Git's SHA-1 to SHA-256 transition, this required translating every historical object ID, leading to years of deployment friction [4]. For append-only audit logs with billions of entries, this is computationally prohibitive and structurally impossible without mutating the history.

We introduce the **Polymorphic Merkle Log (PML)**, an abstract class of cryptographically agile authenticated data structures that resolves this trilemma by maintaining a single unified tree topology shared across multiple hash profiles. We instantiate this via the **Epoch Merkle Log (EML)**, a concrete, RFC 9162-compliant construction where each algorithm is active only during discrete chronological *epochs*. Outside its epochs, an algorithm's projection uses deterministic *null constants*: fixed-point digests domain-separated from leaf and node hashes. A new algorithm is registered at size n in $\mathcal{O}(\log n)$ time without retroactive hashing or forking.

EML eliminates dangerous dynamic negotiation by committing all algorithm epoch boundaries to an append-only global manifest. Because verifiers check proofs against this manifest, any attempt to present a downgraded algorithm configuration or forge inactivity at an active position is immediately detected as a signature mismatch or tree equivocation, securing agility without structural complexity.

The trade-off is explicit: historical data remains bound to the algorithms that were active at the time of insertion. A newly added algorithm verifies only the data appended during its active epochs; pre-epoch positions are committed as deterministic null constants—unforgeable commitments to algorithm inactivity at specific historical positions—that carry no payload verification. This is a principled choice, not an evasion of the retroactive computation cost.

The key theoretical result is *Projection Equivalence*: for any algorithm a, the EML's per-algorithm projection π_a yields a standard RFC 9162 Merkle tree whose leaves are the null-or-real values at each position. This projection is a valid input to unmodified RFC 9162 verification. Correctness of the multi-algorithm EML thus *reduces* to correctness of the single-algorithm standard, a property we prove via machine-checked theorem in §IV.

1.1 Motivating Case Study: Long-Lived Key Directories

Our work is motivated by long-lived key directories and identity registries, which face conflicts between device heterogeneity and decade-scale log lifespans. Resource-constrained IoT hardware may be restricted to hardware-accelerated SHA-256, while servers require SHA3 or BLAKE2 for throughput, and compliance mandates eventually dictate transitions. While parallel single-algorithm trees allow agility at endpoints, they fragment the absolute ordering timeline into separate namespaces, introducing state drift and cross-log synchronization boundaries. The PML resolves this by enforcing a single, universal index space across all projections, unifying timeline auditing without duplicating state.

1.2 Contributions

We make the following contributions:

1. **The Polymorphic Merkle Log (PML) paradigm and EML concrete construction** (§III): we introduce the PML abstract class and its epoch-based concrete instantiation, the EML. The EML maintains a shared-topology append-only Merkle tree supporting concurrent, independent hash algorithms via null-padded chronological epochs. It achieves $\mathcal{O}(1)$ amortized appends per active algorithm, $\mathcal{O}(\log n)$ algorithm addition, and $\mathcal{O}(1)$ algorithm removal.

2. **A machine-checked formal model of the PML algebra** (§IV): 15 definitions and five theorems (Structural Bridge Lemma, Projection Equivalence, Temporal Binding, Algorithm Isolation, and the Generalized Bridge Lemma) mechanized in Lean 4 (~1400 lines, zero `sorry`). The model decouples structural tree arithmetic over the free magma `MerkleTree` α from cryptographic hash function properties, proving topological equivalence and the shift-reduce duality for any append-consistent topology. To our knowledge, this includes the first machine-checked formalization of RFC 9162's MTH construction.

3. **Deterministic proof elision** (§VI): a zero per-proof-metadata wire optimization that compresses inclusion proofs from $\mathcal{O}(\log n)$ to $\mathcal{O}(\log n_a)$ siblings, where n_a is the algorithm's active entry count. The verifier reconstructs elided null siblings via interval arithmetic over the epoch boundaries, requiring no explicit markers or bitmaps.

4. **A production Rust implementation with empirical evaluation** (§VII): 80 tests across unit, property-based, fault injection, and complexity regression categories; 4 fuzz targets with approximately 2 million runs and zero panics; and $\mathcal{O}(\log n)$ complexity confirmed via curve fitting against formal bounds.

1.3 Roadmap

§II establishes the RFC 9162 baseline and design constraints. §III introduces the EML construction informally. §IV presents the formal model. §V describes the operational proof engine. §VI details wire-level proof elision. §VII reports empirical results. §VIII situates the EML within the literature. §IX concludes.

2 Background and Threat Model

2.1 RFC 9162 Merkle Trees

Polymorphic Merkle Logs (PMLs) extend the standard single-algorithm append-only Merkle tree topology defined in RFC 9162 [10] (which we refer to as a **Merkle append-only Log Tree (MALT)** for comparative shorthand), formalizing the construction introduced by Crosby and Wallach [2] and deployed at scale through Certificate Transparency [9]. We present the Epoch Merkle Log (EML) as a concrete instantiation of the PML class. We summarize the relevant definitions.

A log of n leaf entries $D_n = \{d_0, d_1, \ldots, d_{n-1}\}$ is represented by a Merkle Tree Hash (MTH) computed recursively:

$$\mathrm{MTH}(\emptyset) = H(\texttt{""}) \tag{1}$$

$$\mathrm{MTH}(\{d\}) = H(\texttt{0x00} \,\|\, d) \tag{2}$$

$$\mathrm{MTH}(D_n) = H(\texttt{0x01} \,\|\, \mathrm{MTH}(D_{[0:k]}) \,\|\, \mathrm{MTH}(D_{[k:n]})) \tag{3}$$

where k is the largest power of two less than n, and $\|$ denotes byte concatenation. The prefix bytes `0x00` and `0x01` provide domain separation between leaf and internal node

hashes, preventing second-preimage attacks across tree levels [13].

Two proof types verify the log: an **inclusion proof** demonstrates membership of leaf m via $\lceil \log_2 n \rceil$ sibling hashes, and a **consistency proof** demonstrates that the first m entries of an n-leaf tree share the same root as the m-leaf tree, enforcing append-only integrity. Both require $\mathcal{O}(\log n)$ hash operations.

2.2 The Algorithm Binding Constraint

In RFC 9162, the hash function H is a global parameter of the log. Every leaf hash, internal node hash, and root hash is computed under a single algorithm. The specification permits algorithm selection via IANA registries, but this selection is fixed at log initialization. Changing the algorithm requires creating a new log instance—a structural constraint, not merely an operational policy [10].

This binding produces the upgrade trilemma described in §I: any log whose identity is derived from its root hash is, by construction, married to the algorithm that produced that hash. The PML class relaxes this binding by allowing multiple algorithms to coexist over a shared topology, each producing an independent root.

2.3 Random Oracle Constraint

We operate under the Random Oracle Model (ROM) for all hash functions. Specifically, we require:

- **Collision resistance.** It is computationally infeasible to find distinct inputs $x \neq y$ such that $H(x) = H(y)$.
- **Preimage resistance.** Given a hash output h, it is computationally infeasible to find any input x such that $H(x) = h$.
- **Cross-algorithm independence.** For distinct algorithms $a \neq b$, the output distributions of H_a and H_b are mutually incompressible: knowledge of $H_a(x)$ provides no information about $H_b(x)$. Under ROM, this follows from the independence of distinct random oracles.
- **Domain separation.** For any hash function H_a, inputs with distinct prefix tags produce computationally independent outputs. Specifically, $H_a(\texttt{0x00} \,\|\, d) \neq H_a(\texttt{0x02})$ for all d—a collision between the leaf and null domains requires breaking preimage resistance of H_a. This is the sole cryptographic assumption required by Temporal Binding (Theorem 2, §IV).

The third property is the cryptographic foundation for Algorithm Isolation (Theorem 3, §IV): adding a weak algorithm to the EML cannot weaken a strong one, because the hash outputs of distinct algorithms are informationally independent.

Table 1: Design constraints governing the EML construction.

ID	Constraint	Rationale
C1	Append-only. No mutation of committed data.	Foundational integrity property.
C2	Multiple algorithms over a shared topology.	Avoid per-algorithm tree duplication.
C3	Zero retroactive computation on algorithm addition.	$\mathcal{O}(N)$ rehashing is unacceptable for large logs.
C4	Algorithm-independent verification.	No weakest-link coupling between algorithms.
C5	Standard-compatible verification core.	The cryptographic verification function (hash-chain walk) must be unmodified RFC 9162. Elision-aware clients perform a rehydration step before invoking this core.
C6	Algorithm lifecycle management.	Algorithms activate, deactivate, and resume over time.
C7	Deterministic state.	All values computable from the log's append history and algorithm metadata.

2.4 Design Constraints

The EML is designed under seven constraints, each motivated by the requirements of long-lived verifiable logs:

2.5 Threat Model

The adversary controls the network between the log server and verifying clients. They can delay, reorder, or drop messages but cannot forge hash preimages or collisions under any algorithm satisfying the ROM assumption. The adversary may also control the log server itself, attempting to present inconsistent views (state-forking equivocation) to different clients, or silently substitute historical payloads (second-preimage attacks) if underlying hash function security decays. EML inherits the same consistency guarantees as RFC 9162: any two clients that agree on a tree size and root hash for a given algorithm are guaranteed to observe identical history. Importantly, these guarantees treat state-forking equivocation and historical payload substitution as distinct cryptographic failure modes, as detailed below.

The adversary *cannot* exploit the presence of multiple algorithms to weaken any single algorithm's guarantees. This is the content of Algorithm Isolation (Theorem 3, §IV): each

projection is informationally independent under ROM.

Why not independent trees? An alternative design maintains entirely separate Merkle trees per algorithm, binding their roots via a composite STH. This avoids null padding but sacrifices the fundamental property of a Merkle append-only log: a single, cryptographically committed chronological ordering. In the EML, position i is position i for every algorithm—some contribute a leaf hash, some contribute a null constant, but the temporal ordering is universal and structurally committed. Independent trees fragment this chronology into $|\mathscr{A}|$ separate orderings. A composite STH snapshots the state of each tree at signing time but commits nothing about the temporal interleaving between trees: it cannot express that entry j in tree a and entry k in tree b were contemporaneous, nor can it represent the absence of algorithm a at a particular point in the system's history.

The EML's null constants are not inert padding; they are **unforgeable witnesses of algorithm inactivity**. A null at position i for algorithm a is a committed statement—embedded in the Merkle hash chain, reflected in $\text{root}(a)$, signed by the STH—that algorithm a was inactive at the i-th event. To forge activity at a position where none existed, an attacker must either: (1) alter the leaf representation from $N_0(a)$ to a valid data leaf $\text{leaf}(a,d)$, which necessarily changes $\text{root}(a)$ and requires the malicious log to sign a distinct, conflicting STH to cover its tracks (detected as state-forking equivocation via verifier gossip); or (2) execute a second-preimage attack by finding a payload d such that $\text{leaf}(a,d) = N_0(a)$, which preserves the root hash and thus bypasses equivocation detection entirely. Under the ROM assumption, the latter is computationally infeasible. Independent trees provide no structural representation of inactivity and thus no mechanism to cryptographically prove it. This temporal witness operates at algorithm granularity; the tree commits which algorithm was active at each position, while specific payload identity is resolved by the leaf payload itself.

Operationally, the shared topology also yields unified consistency proofs (a single check vs. $|\mathscr{A}|$ independent checks) and shared proof paths for multi-algorithm verification. The null-padding storage overhead is $\mathscr{O}(|\mathscr{A}| \cdot \log n)$ in NullTable entries, not $\mathscr{O}(|\mathscr{A}| \cdot n)$: null subtree hashes are computed on the fly from the precomputed NullTable, not stored per-position.

Historical verification and security tradeoffs. The EML does not retroactively verify historical data under newly added algorithms. A newly activated algorithm verifies only the data appended during its active epochs; pre-epoch positions are committed as deterministic null constants that are unforgeable but carry no payload verification (Definition 4.3). This design reflects three fundamental security and operational tradeoffs:

1. *Preimage vs. Collision Asymmetry:* Cryptographic hash functions typically succumb to collision attacks (enabling future forgeries) decades before they succumb to preimage attacks (enabling modification of past history). The EML protects future appends by immediately transitioning to a collision-resistant algorithm, while relying on the remaining preimage resistance of the deprecated algorithm to protect the past.
2. *Operational Viability:* The alternative is an $\mathscr{O}(N)$ retroactive rehashing migration, which is computationally prohibitive and structurally impossible for live immutable logs without downtime or violating write invariants. EML enables $\mathscr{O}(1)$ instant algorithm additions.
3. *History Preservation vs. Equivocation Detection:* If an adversary succeeds in finding a second preimage for a deprecated algorithm, they can silently substitute a historical payload without altering its root hash. Because this leaves the Signed Tree Head (STH) unchanged, transparency monitors observe zero state changes, providing zero defense-in-depth against payload substitution once second-preimage resistance falls. History preservation thus relies strictly on the remaining preimage resistance of the deprecated primitive itself. The monitor ecosystem's equivocation detection only protects against history modification under active, preimage-resistant algorithms, where any payload modification alters the root, forcing the log to sign distinct STHs to different clients to avoid detection.

Epoch manifest commitment via the Signed Tree Head. The EML binds the activation manifest directly to the log's root of trust. The Signed Tree Head (STH) commits the global tree size, and per-algorithm roots, tree sizes, and cryptographic digests of each algorithm's canonical activation map (formalized as Definition 8 in §IV). This binding guarantees that any attempt by a malicious log to present conflicting manifests to different clients requires signing distinct STHs—a classical equivocation detectable by standard client-gossip and witness protocols. The manifest commitment also closes the shared-state assumption underlying proof elision (§VI): agreement on the STH implies agreement on the epoch topology (Corollary 1 (Manifest Commitment)), so verifiers need not trust any out-of-band channel for epoch metadata.

Equivocation detection. Cross-algorithm equivocation—a malicious log forking algorithm a's history while presenting a consistent history under algorithm b—is detectable because the STH signs all per-algorithm roots atomically. Forking π_a forces forking the global STH, and monitors on any π_b will detect the equivocation. Formally: equivocation on any projection π_a is detectable if at least one honest monitor performs consistency checks on the unified STH.

3 Design Paradigm and Architectural Topology

This section introduces the Polymorphic Merkle Log (PML) paradigm and its concrete Epoch Merkle Log (EML) con-

struction informally. The formal algebraic model follows in §IV.

3.1 Shared Topology

A PML is a single binary Merkle tree. Every registered algorithm observes the same tree topology—the same number of leaf positions, the same branching structure, the same internal node relationships (as illustrated in Figure 1). What differs across algorithms is the *value* computed at each node: each algorithm hashes every position independently using its own hash function.

This is not one tree per algorithm. It is a single, unified data structure with multiple projections. While storing distinct internal node hashes for each algorithm is storage-equivalent to parallel logs, this shared-topology model yields three structural advantages over running disjoint logs:

1. **Timeline Continuity and Shared Sequencing:** The log enforces a single, globally ordered index space. Index n under algorithm a represents the exact same entry and transaction timeline as index n under algorithm b. This eliminates coordinate fragmentation, allowing applications to reason about strict, chronological ordering across different cryptographic epoch transitions.
2. **Atomic Write Coordination:** A single append operation updates the logical state of the log atomically across all active projections, ensuring they can never diverge or drift out of sync.
3. **Consolidated Auditing Complexity:** Auditors and monitors watch a single log identity. Equivocation on any projection is immediately detectable as a split-view of the same shared logical sequence, bypassing the complexity of reconciling separate, asynchronous log timelines.

3.2 Null Constants

When a new algorithm a is added to a log that already contains n entries, the algorithm must account for positions 0 through $n-1$ without re-hashing any historical data. The EML accomplishes this through *null constants*: per-algorithm, per-height deterministic values that occupy inactive positions.

The null leaf constant for algorithm a is:

$$N_0(a) = H_a(\texttt{0x02}) \tag{4}$$

The prefix byte `0x02` is distinct from both the leaf prefix `0x00` and the node prefix `0x01`, establishing a third domain through the same single-byte mechanism used by RFC 9162. No additional payload is required—domain separation is achieved by the prefix alone.

Null constants propagate upward through the tree deterministically. A subtree of 2^h null leaves has root:

$$N_h(a) = H_a(\texttt{0x01} \,\|\, N_{h-1}(a) \,\|\, N_{h-1}(a)) \tag{5}$$

Because every null leaf is identical, null subtrees are perfectly symmetric. The entire table $\{N_0(a), N_1(a), \ldots, N_H(a)\}$ is precomputable in $\mathcal{O}(H)$ hash operations, where $H = \lceil \log_2 n \rceil$.

3.3 Leaf Value Function

The value at tree position i for algorithm a is:

$$V(a,i) = \begin{cases} H_a(\texttt{0x00} \,\|\, \text{data}[i]) & \text{if } a \text{ is active at } i \\ N_0(a) & \text{otherwise} \end{cases} \tag{6}$$

This is the central definition. Active positions receive the standard RFC 9162 leaf hash of the raw data. Inactive positions receive the null constant. The tree topology is dense—every position is filled—but the *semantic content* is sparse in the temporal dimension for newly added algorithms.

3.4 Epochs

An algorithm's lifetime within the log is partitioned into *epochs*: disjoint chronological intervals $[[s_1, e_1), [s_2, e_2), \ldots]$ during which the algorithm actively hashes appended data. Between epochs, the algorithm is frozen: its root and frontier state are immutable, and new appends produce null constants in its projection.

Three lifecycle operations govern epochs:

- `add_alg`(a): Registers algorithm a with initial epoch $[n, \infty)$, where n is the current tree size. The algorithm's frontier is initialized to the *null prefix peaks*: the Merkle mountain range (MMR) peaks of n null leaves, each computable as a NullTable lookup. Cost: $\mathcal{O}(\log n)$.
- `remove_alg`(a): Closes a's current epoch at the current tree size. The algorithm's frontier and root are frozen; future appends do not update them. Cost: $\mathcal{O}(1)$.
- `resume_alg`(a): Reopens a frozen algorithm by appending a new epoch $[n, \infty)$. The frontier stack is reconstructed for the current tree size n by decomposing n into its binary representation and resolving each complete subtree root—via stored nodes, the NullTable, or recursive binary splits for mixed boundary subtrees. Cost: $\mathcal{O}(\log n)$. Resume provides operational flexibility to temporarily freeze inactive algorithms (saving write I/O and database storage during quiet periods) and resume them later if verification demand or compliance mandates change, paying a one-time $\mathcal{O}(\log n)$ reconstruction cost.

3.5 One Tree, Many Projections

The *projection* of the PML onto algorithm a yields a sequence of leaf values:

$$\pi_a(\text{PML}) = [V(a,0),\ V(a,1),\ \ldots,\ V(a,\text{tree\_size}(a)-1)]$$

This sequence is a valid input to the standard RFC 9162 MTH construction. The root computed from this projection equals the root maintained by the EML's incremental frontier for algorithm a. From the perspective of a verifier holding a single algorithm's root hash and proof, the EML is indistinguishable from a standard single-algorithm Merkle tree.

Other algorithms are invisible. SHA-256's proofs never reference BLAKE3's hashes. BLAKE3's proofs never reference SHA-256's. The algorithms share topology but not trust.

3.6 Non-Negotiable Agility and Downgrade Resistance

Traditional cryptographic agility designs (such as those in TLS cipher suite negotiation or JSON Web Tokens) rely on dynamic, runtime algorithm negotiation. This dynamic negotiation introduces a significant attack surface: an adversary capable of altering network packets can intercept the handshake and inject weaker algorithms, forcing a client to downgrade its cryptographic security level.

The EML design neutralizes downgrade attacks by completely eliminating runtime algorithm negotiation. Instead of relying on client-server negotiation during verification, the EML binds algorithm lifecycles directly to the log's root of trust via the epoch manifest. This manifest (Definition 8, §IV) is committed in the Signed Tree Head (STH) signed by the log operator. Because client verification enforces that the active epochs for any algorithm match this immutable manifest, any attempt by an adversary to present a downgraded algorithm configuration or forge inactivity at an active position is immediately detected as a signature mismatch or tree-head equivocation.

4 Formal Model

We formalize the abstract algebra of Polymorphic Merkle Logs (PML), its concrete Epoch Merkle Log (EML) instantiation, and state five machine-checked theorems establishing their correctness. The model decouples the tree topology from cryptographic hashing by defining tree arithmetic over the free magma `MerkleTree` α, projecting to concrete cryptographic digests via a unique algebra homomorphism `eval`. Definitions inherited from RFC 9162 are referenced inline. The full proof artifact (Lean 4, ~1400 lines, 31 theorems, zero `sorry`) accompanies this paper.

4.1 Structural Merkle Tree Algebra

To isolate the tree's topological properties from cryptographic hashing, we model the tree algebra over a free magma.

Definition 1 *(Structural Tree & Batch MTH). Let α be an alphabet. A structural Merkle tree $T \in \mathit{MerkleTree}\ \alpha$ is defined inductively as $\mathit{empty} \mid \mathit{leaf}(v) \mid \mathit{node}(l,r)$ for $v \in \alpha$. The batch Merkle Tree Hash $\mathit{mth}(L)$ maps a leaf sequence L of length n to a structural tree: empty if $n = 0$, d_0 if $L = [d_0]$, and $\mathit{node}(\mathit{mth}(L[0:k]), \mathit{mth}(L[k:n]))$ if $n > 1$, where $k = \mathit{largestPow2Lt}(n)$ is the largest power of two strictly less than n (§II).*

Definition 2 *(Incremental CTO Accumulator). An incremental forest is represented by a stack $S \in \mathit{List}(\mathit{MerkleTree}\ \alpha)$ of descending power-of-two subtree heights. Appending a leaf ℓ at index n is defined by carry-propagation: $\mathit{appendToStack}(S, \ell, n) = \mathit{mergeStack}(\ell :: S, \mathit{cto}(n))$, where $\mathit{cto}(n)$ is the number of trailing one-bits of n (carry height), and $\mathit{mergeStack}(r :: l :: \mathit{rest}, c+1) = \mathit{mergeStack}(\mathit{node}(l,r) :: \mathit{rest}, c)$ (with underflow fallback). The incremental root is $\mathit{ctoRoot}(L) = \mathit{stackRoot}(\mathit{foldl}(\lambda(S,(\ell,i)).\ \mathit{appendToStack}(S,\ell,i), [], L))$, where $\mathit{stackRoot}(h :: t)$ folds the forest top-to-bottom: $\mathit{foldl}(\lambda(\mathit{acc},\ell).\ \mathit{node}(\ell,\mathit{acc}), h)$.*

4.2 Cryptographic Homomorphism

The structural tree is mapped to a concrete cryptographic digest space via a unique homomorphism.

Definition 3 *(Cryptographic Homomorphism). Let Digest be a digest space. A cryptographic hash function $H : \mathit{Vec}\langle \mathit{UInt8} \rangle \to \mathit{Digest}$ evaluates structural nodes under domain separation tags: $\mathit{leaf}(v) = H(\texttt{0x00} \parallel v)$, $\mathit{node}(l,r) = H(\texttt{0x01} \parallel l \parallel r)$, and $\mathit{empty} = H(\texttt{""})$. The unique algebra homomorphism $\mathit{eval} : \mathit{MerkleTree}(\mathit{Digest}) \to \mathit{Digest}$ commutes with these constructors (e.g., $\mathit{eval}(\mathit{node}(l,r)) = H(\texttt{0x01} \parallel \mathit{eval}(l) \parallel \mathit{eval}(r))$). We define $\mathit{mthDigest}(L) = \mathit{eval}(\mathit{mth}(L.\mathit{map}(\mathit{leaf})))$ and $\mathit{ctoRootDigest}(L) = \mathit{eval}(\mathit{ctoRoot}(L.\mathit{map}(\mathit{leaf})))$.*

4.3 EML Topology and Lifecycles

EML builds on this homomorphic separation to manage multi-algorithm epochs.

Definition 4 *(Null constants). For each algorithm a, the null leaf constant is $N_0(a) = H_a(\texttt{0x02})$. The null subtree constant of height $h \geq 1$ is defined recursively:*

$$N_h(a) = \mathit{node}_a(N_{h-1}(a),\ N_{h-1}(a))$$

The null constant table $\{N_0(a), \ldots, N_H(a)\}$ is precomputable in $\mathcal{O}(H)$ hash operations.

Definition 5 *(Activation Map). An activation map is a partial function $\mathit{act} : \mathit{Alg} \rightharpoonup \mathit{Vec}\langle [\mathbb{N}, \mathbb{N} \cup \{\infty\}) \rangle$ mapping each algorithm to disjoint chronological epochs. Algorithm a is active at index i, written $\mathit{active}(a,i)$, iff $\exists [s_k, e_k) \in \mathit{act}(a)$ such that $s_k \leq i < e_k$.*

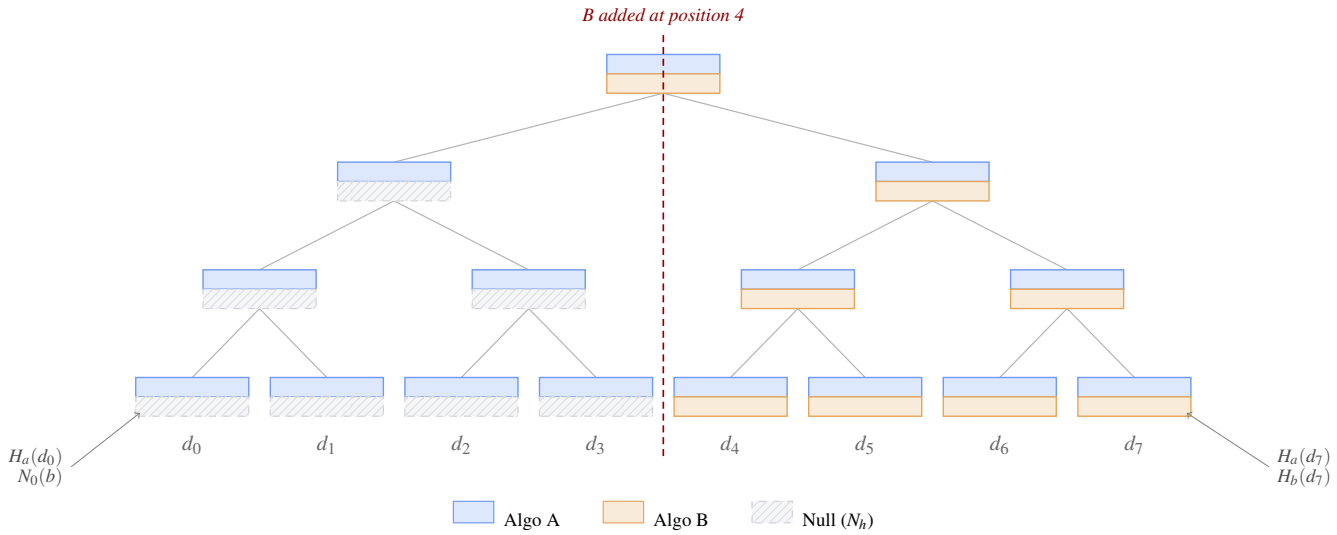

Figure 1: An 8-leaf PML with two algorithms. Algorithm A (blue) is active from genesis. Algorithm B (amber) was added at position 4; its pre-activation positions are filled with null constants (gray, dashed). Each node computes independent hash values per algorithm over a shared topology.

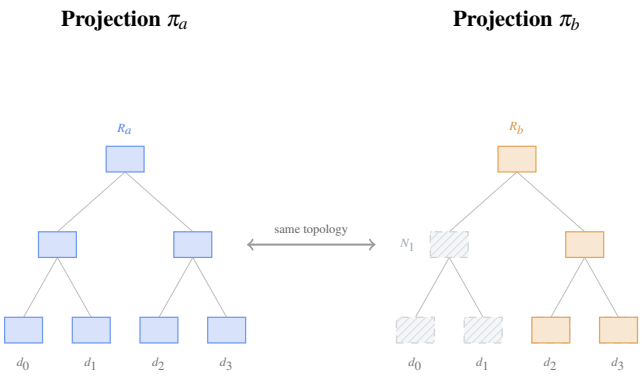

Figure 2: Projection view of the PML from Figure 1. Left: Algorithm A's projection — a standard Merkle tree with all positions active. Right: Algorithm B's projection — positions 0–3 are null constants (N_h), positions 4–7 are active. Each projection is an independently valid Merkle tree; proofs are generated per-projection.

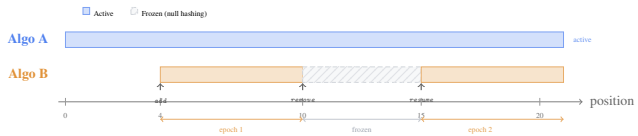

Figure 3: Algorithm lifecycle in the EML. Algorithm A is active from genesis. Algorithm B is added at position 4, removed at position 10 (entering a frozen state where null hashing continues), and resumed at position 15. Each active interval defines an *epoch*; the frozen gap preserves tree-wide append consistency.

Definition 6 *(Leaf Value & State). The EML state is a tuple* $S = (leaves, size, act, stacks, nodes)$ *where leaves is the raw payload vector,* $stacks(a)$ *is the forest stack for a, and* $nodes(a)$ *maps subtree ranges to sealed node hashes. The digest at position i for algorithm a is* $V(a,i) = H_a(\texttt{0x00} \,\|\, leaves[i])$ *if* $active(a,i)$ *and* $N_0(a)$ *otherwise.*

Definition 7 *(EML Operations). Appending payload d updates raw leaves* $S'.leaves = S.leaves +\!\!+ [d]$ *and, for each active algorithm a,* $stacks(a) = appendToStack(S.stacks(a), leaf(V(a,S.size)), S.size)$. *Frozen (inactive) algorithms are not updated. -* `add_alg`(a)*: sets* $act(a) = [(S.size, \infty)]$ *and initializes the stack to the* null prefix peaks $[leaf(N_{h_i}(a)) \mid$ *bit i set in* $S.size$*, descending*$]$. *-* `remove_alg`(a)*: closes a's last epoch at* $S.size$*. -* `resume_alg`(a)*: opens epoch* $[S.size, \infty)$ *and reconstructs the stack by decomposing* $S.size$ *and resolving each perfect subtree root in* $\mathcal{O}(\log S.size)$ *via stored nodes,* $N_h(a)$*, or recursive binary splits.*

Definition 8 *(Root and Signed Tree Head). The root of algorithm a is* $root(a) = H_a(\text{""})$ *if* $stacks(a) = []$ *and* $eval(stackRoot(stacks(a)))$ *otherwise. The Signed Tree Head (STH) is:*

$$STH = Sign_\sigma\big(n,\ t,\ \{\,(a,\ root(a),\ tree\_size(a),\ H_a(act(a)))\ \mid a \in dom(act)\,\}\big)$$

where $H_a(act(a))$ *is the cryptographic digest of a's activation map. The map is canonically serialized as a 64-bit big-endian length prefix specifying the epoch count, followed by a contiguous sequence of 16-byte blocks, where each block contains an 8-byte big-endian* `start` *index and an 8-byte big-endian*

end index (using $2^{64}-1$ as the active sentinel).

Definition 9 *(Projection). The projection of the EML onto algorithm a is the leaf digest sequence $\pi_a(S) = [V(a,i) \mid 0 \leq i < tree\_size(a)]$.*

4.4 Generalized Shift-Reduce Duality

We demonstrate that the structural compatibility of EML with standard log operations is not an ad-hoc bitwise artifact of standard binary configurations. Rather, it is a manifestation of a deeper combinatorial property: a generalized shift-reduce duality that holds for right-leaning, left-decomposing append-consistent tree layouts. By formalizing this equivalence over generalized split policies and merge schedules, we establish that structural correctness is a universal invariant of right-leaning tree decompositions.

Definition 10 *(Generalized Duality & Append-Consistency). A split policy $f : \mathbb{N} \to \mathbb{N}$ satisfies $0 < f(n) < n$ for all $n > 1$. A merge schedule $s : \mathbb{N} \to \mathbb{N}$ indicates stack reductions at step n. Let $forestSizes(f,n)$ define the right-leaning forest layout in the stack at size n: $[]$ if $n = 0$, $[1]$ if $n = 1$, and $forestSizes(f,n-k) +\!\!+ [k]$ if $n > 1, k = f(n)$. The pair (f,s) is* append-consistent *iff for all n: $forestSizes(f,n+1) = (1+\sum take(c,S_n)) :: drop(c,S_n)$ (where $c = s(n)$ and $S_n = forestSizes(f,n)$), and for all $i < c$, the split point of the merged subtree matches the policy: $f(1+\sum take(i+1,S_n)) = S_n[i]$.*

4.5 Theorems

The following theorems are machine-checked in Lean4.

Theorem 1 *(Structural Bridge Lemma). For any leaf sequence ℓ, incremental root extraction is topologically identical to batch hashing: $ctoRoot(\ell) = mth(\ell)$.*

Proof sketch. Proved by strong induction on length, showing that appendToStack preserves a descending power-of-two forest layout. The core induction step relies on a *descent condition* (`cto_trailing_geo`): when an append triggers c carry merges, the top c segments must form a geometric series $2^0, \ldots, 2^{c-1}$ to be merged. A modular arithmetic contradiction proof in Lean 4 (~180 lines) establishes that a degenerate segment layout would require a carry propagation larger than the one that actually occurred, closing the edge-case boundary errors that paper proofs typically elide. □

Theorem 2 *(Projection Equivalence). For any algorithm a, the EML's incremental root equals the batch root of its projected leaf sequence: $root(a) = mthDigest(\pi_a(S))$.*

Proof. Follows by applying the unique algebra homomorphism `eval` (Definition 3) to both sides of Theorem 1: $\text{eval}(\text{ctoRoot}(\ell)) = \text{eval}(\text{mth}(\ell))$. Because `eval` is a homomorphism, it commutes with the structural constructors, establishing topological equivalence over any concrete cryptographic digest space in a single step (`generic_projection_equivalence`) without assuming collision resistance or hash output distributions. □

Theorem 3 *(Temporal Binding). For any algorithm a and inactive position $i < tree\_size(a)$, there is no payload $d \in Bytes$ such that $V(a,i) = leaf(a,d)$.*

Proof. In the formal model, this is checked via the structural domain separation axiom `nullLeaf != leafHash d` (representing $N_0(a) \neq \text{leaf}(a,d)$). In the physical implementation, this structural disjointness maps to a *computational hardness assumption under the Random Oracle Model (ROM)*. Since the inputs to the hash function H_a are prefixed with distinct domain separation tags (`0x02` for the null constant and `0x00` for leaf data), finding a payload d that violates this inequality is equivalent to finding a hash collision between different domain prefixes. Under the ROM, the probability of an adversary finding such a collision or preimage is negligible, bounding the security of the temporal witness to the hash function's cryptographic strength. Mechanized in Lean 4 as `Ne.symm (domain_separation d)`. □

Theorem 4 *(Algorithm Isolation). Projections for different algorithms a and b operating on the same log are structurally independent: $root(a) = mthDigest(\pi_a(S)) \wedge root(b) = mthDigest(\pi_b(S))$.*

Proof. Each conjunct reduces to an instance of Theorem 2. Mechanized in Lean 4 as `⟨bridge_lemma _, bridge_lemma _⟩`. Because neither conjunct references the other's epochs or hashing primitives, algorithm transitions cannot degrade the security guarantees of other projections. □

Theorem 5 *(Generalized Bridge Lemma). For any split policy f and merge schedule s that are `AppendConsistent` (Definition 10), the generalized incremental stack machine equals the generalized right-leaning batch tree: $generalized\_ctoRoot(s,\ell) = generalized\_mth(f,\ell)$.*

Proof sketch. Proved by showing that the compatibility relation `AppendConsistent` transitions stack sizes coherently under $s(n)$ merges, and that folding the forest top-to-bottom combines subtrees in the exact bisection order mandated by policy f. Proved purely combinatorially in `Duality.lean` (`generalized_bridge_lemma`). The linear split policy ($f(n) = 1$) and trivial schedule ($s(n) = 0$) are proven compatible in `Instantiation.lean` (representing a simple right-leaning hash chain), verifying that the generalized framework admits concrete instances without cryptographic axioms. □

The operational significance of Theorem 5 cannot be overstated: it demonstrates that algorithm-independent log agility is a universal structural property of right-leaning, left-decomposing append-consistent tree topologies. By decoupling split policies from concrete carry masks, the PML paradigm provides a blueprint for formalizing lifecycle trans-

formations across alternative data structures that utilize left-to-right append-only layouts—including right-leaning Sparse Merkle Trees and Merkle Patricia Tries—guaranteeing that structural correctness can be checked independently of any cryptographic mechanism.

4.6 Corollaries & Proof Artifact

The EML's correctness properties reduce to Theorem 2 combined with standard RFC 9162 guarantees: **Inclusion/Consistency soundness** (standard RFC 9162 verifiers accept EML proofs) and **Projection validity** ($\pi_a(S)$ yields a standard RFC 9162 leaf sequence).

Corollary 4.1. ***Corollary 1*** *(Manifest commitment). For any two clients C_1, C_2 that accept the same Signed Tree Head (STH), their activation maps agree: $act_{C_1} = act_{C_2}$.*

Agreement on the STH cryptographically guarantees agreement on the epoch boundaries, ensuring clients share the same epoch boundaries for proof elision (§VI) without out-of-band assumptions.

The Lean 4 proof artifact (~1400 lines, 31 theorems, zero `sorry`) consists of two parallel tracks: the concrete path verifying RFC 9162/CTO equivalence, and the generalized path proving shift-reduce duality. It is the first machine-checked formalization of RFC 9162's MTH construction in any proof assistant, serving the Certificate Transparency ecosystem broadly.

5 Proof Generation and Verification

The formal model (§IV) defines the EML's mathematical structure. We now describe the operational proof engine: how proofs are generated in $\mathcal{O}(\log n)$ time without materializing the full projection.

5.1 The Subtree Root Query

The projection mapping $\pi_a(S)$ defines the system's external mathematical specification. The operational proof engine achieves $\mathcal{O}(\log n)$ execution by resolving sibling hashes through coordinate point lookups directly over the physical storage traits.

Definition 11 *(Subtree root). The root hash of the subtree covering leaves $[lo, hi)$ for algorithm a:*

$$sr(a, lo, hi) = \begin{cases} empty(a) & w = 0 \\ V(a, lo) & w = 1 \\ N_h(a) & \text{null range},\ w = 2^h \\ nodes(a)[(lo, h)] & \text{stored},\ w = 2^h \\ node_a(sr(a, lo, l'), sr(a, l', hi)) & \text{otherwise} \end{cases}$$

where $w = hi - lo$, $l' = lo + k$, $k = largestPow2Lt(w)$, and $h = \log_2 w$.

Table 2: Operational complexity bounds. K: tree size at algorithm addition; n: global tree size; $|\mathscr{A}|$: registered algorithm count.

Operation	Complexity	Mechanism
Append (per algorithm)	$\mathcal{O}(1)$ amortized	CTO merge + node persistence
Append (total)	$\mathcal{O}(\lvert A(i)\rvert)$ amortized	Linear in active algorithm count
Algorithm addition	$\mathcal{O}(\text{popcount}(K))$ worst-case	Null prefix peak computation
Algorithm resumption	$\mathcal{O}(\log n)$	Frontier reconstruction via `subtree_root`
Root extraction	$\mathcal{O}(\log n)$	Frontier stack fold
STH construction	$\mathcal{O}(\lvert\mathscr{A}\rvert \cdot \log n)$	Root extraction per algorithm + manifest hash
Inclusion proof	$\mathcal{O}(\log n)$	`subtree_root` resolution
Consistency proof	$\mathcal{O}(\log n)$	`subtree_root` resolution

Definition 12 *(Active range). Algorithm a has active content in $[lo, hi)$ iff any epoch overlaps it: $active\_range(a, lo, hi) \iff \exists [s_k, e_k) \in act(a).\ s_k < hi \wedge e_k > lo$.*

The cases of Definition 11 are prioritized: (1) empty range, (2) single leaf, (3) null subtree (overlaps no epochs, resolved via `NullTable` in $\mathcal{O}(1)$ for power-of-two, or $\mathcal{O}(\log w)$ recursive splits), (4) stored node (resolved via database lookup in $\mathcal{O}(1)$), and (5) recursive bisection split.

5.2 Complexity & Storage Efficiency

Each recursive call to `subtree_root` either terminates via a stored node or `NullTable` lookup ($\mathcal{O}(1)$), or splits into two subproblems where at least one child terminates. The proof path requires at most $\lceil \log_2 n \rceil$ sibling resolutions, each requiring a single `subtree_root` invocation. Total complexity is $\mathcal{O}(\log n)$ lookups and hash operations (see Table 2).

Storage and Database I/O Sparsity. EML does not persist null leaves or null internal nodes in the storage backend. When an algorithm is frozen, it is excluded from appending, saving write I/O. During proof generation, `subtree_root` intercepts null ranges $[lo, hi)$ in-memory using the local `NullTable` without triggering database queries. Only nodes whose intervals intersect active epochs require physical storage, minimizing the I/O footprint.

5.3 Inclusion and Consistency Proofs

Proofs follow the RFC 9162 `PATH` and `SUBPROOF` algorithms exactly, with `subtree_root` replacing materialized-array slicing at each sibling resolution.

Definition 13a *(Inclusion Proofs). The inclusion proof is* $\textit{inclusion\_proof}(S,a,m) = \textit{path}(S,a,m,0,\textit{tree\_size}(a))$ *where path collects sibling hashes via subtree_root.*

Definition 13b *(Consistency Proofs). The consistency proof is* $\textit{consistency\_proof}(S,a,m) = \textit{subproof}(S,a,m,0,\textit{tree\_size}(a),\textit{true})$ *where* `subproof` *follows RFC 9162 §2.1.4.*

Consistency proofs operate correctly across epoch boundaries because the `subproof` algorithm decomposes the tree by index ranges, which `subtree_root` resolves transparently via stored nodes, the `NullTable`, or recursive bisection. By Theorem 2, the proofs verify against the STH root.

5.4 Node Caching, Frontier & Cold Reconstruction

During CTO merges in `append`, each merge of two stack entries produces a parent hash written to the node store keyed by `(algorithm, left_index, height)`. Null subtrees are never written to disk.

For restarts, the `from_storage` constructor reconstructs the EML state in $\mathcal{O}(\log n)$ per algorithm by loading epoch metadata and resolving the peaks of the tree size n through the node store. During algorithm resumption (`resume_alg`), subtrees that overlap the boundary between active data and the null gap produce *mixed nodes* that cannot be synthesized from the `NullTable` alone. EML eagerly collects these mixed boundary nodes (at most $\lceil \log_2 n \rceil$ nodes) during frontier reconstruction and commits them to the node store in a single transaction via the atomic batch storage API (`write_batch`). This prevents database I/O bottlenecks and guarantees that subsequent query operations traversing the gap boundary find nodes directly in the store, preserving the strict $\mathcal{O}(\log n)$ complexity without recursive split overhead.

6 Wire Optimization: Elided Proofs

The EML's shared topology introduces a proof depth of $\mathcal{O}(\log n)$, compared to $\mathcal{O}(\log n_a)$ in independent single-algorithm logs of size n_a. When $n_a \ll n$, proofs contain siblings covering ranges where algorithm a was inactive. Since these siblings consist entirely of null constants, they are deterministically reconstructable by any client possessing the epoch boundaries, enabling us to compress wire size to $\mathcal{O}(\log n_a)$.

6.1 Null Sibling Detection & Elision Protocol

For algorithm a, leaf index m, and tree size n, each sibling covers a range $[lo, hi)$. The sibling is null if $\neg\text{active\_range}(a, lo, hi)$. The protocol operates as:

1. **Generation (server).** The server generates the full RFC 9162 inclusion proof path containing $\lceil \log_2 n \rceil$ siblings (§V).
2. **Elision (server).** For each sibling, if $\neg\text{active\_range}(a, lo, hi)$, the sibling is omitted from the wire. The resulting *elided proof* contains only active-epoch siblings.
3. **Rehydration (client).** The client walks the virtual tree path for index m. If a sibling's range $[lo, hi)$ of width w is inactive, the client synthesizes its hash locally from its `NullTable` in $\mathcal{O}(1)$ if $w = 2^h$ (using $N_h(a)$), or in $\mathcal{O}(\log w)$ recursive splits otherwise (Definition 11). If active, the client consumes the next sibling from the elided proof wire payload.
4. **Verification (client).** The rehydrated proof is passed to the unmodified RFC 9162 cryptographic verification engine.

6.2 Zero Per-Proof Metadata

No markers or bitmaps are transmitted to signal elisions. Prover and verifier independently compute identical elision decisions using only the shared state: the tree size n, leaf index m, and the epoch boundaries $\text{act}(a)$ (committed to the STH, see Corollary 1).

However, this shifts the bandwidth cost to a one-time session-level bootstrap payload: clients must download the activation map act. For $|A|$ algorithms with an average of E epochs, the canonical serialization (Definition 8) yields a bootstrap payload of exactly $|A| \cdot (8 + 16 \cdot E)$ bytes. For a typical deployment of $|A| = 4$ algorithms with $E = 2$ epochs, this results in a negligible one-time transfer of 160 bytes. Because algorithm lifecycle operations are rare (occurring on the order of months or years), this tiny cost is amortized over millions of subsequent proofs, resulting in a net wire overhead of zero per proof. This contrasts with SMT compression, which requires transmitting an explicit bitmap (typically 32 bytes) with *every* proof [8, 19], incurring a higher cumulative cost after only 5 proofs.

6.3 Wire Size Analysis

Let n be the tree size and n_a the number of active positions for algorithm a. For a single epoch $[s, \infty)$, siblings covering the null prefix $[0, s)$ are elided. The elided proof contains approximately $\lceil \log_2 n_a \rceil$ siblings. For multiple disjoint epochs, the wire size is bounded by $\mathcal{O}(\log n_a + k)$ where k is the number of epoch boundaries intersecting sibling coverage ranges.

Table 3: Wire proof size comparison. n: tree size; n_a: active entries for the queried algorithm; N_{items}: SMT non-empty entries. * EML elided proofs carry zero per-proof metadata, but require a session-level bootstrap payload of $\mathcal{O}(|A| \cdot E)$ for activation maps.

Proof system	Wire size	Metadata overhead
RFC 9162 (standard)	$\mathcal{O}(\log n)$ hashes	None
SMT (bitmap compressed)	$\mathcal{O}(\log N_{\text{items}})$ hashes	32-byte bitmap
EML (full proof)	$\mathcal{O}(\log n)$ hashes	None
EML (elided proof)	$\mathcal{O}(\log n_a)$ hashes	**None***

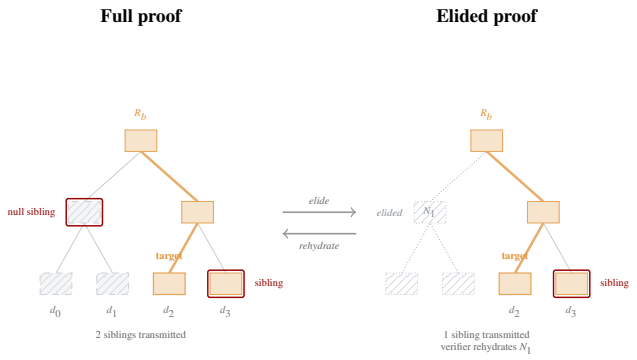

Figure 4: Proof elision. Left: a full inclusion proof for Algorithm B at position d_2 requires two siblings, including a null subtree root covering positions 0–1. Right: the elided proof omits the null sibling entirely; the verifier rehydrates it from the precomputed null table $N_h(b)$ using only the epoch boundary metadata.

The EML's elided proofs achieve comparable compression to SMTs but without transmitting metadata.

7 Implementation and Evaluation

Our evaluation demonstrates that the EML systematically breaks the upgrade trilemma, delivering complete cryptographic agility as an inherent architectural property with zero runtime penalty relative to single-algorithm deployments. We implement the EML as a Rust crate providing a generic, storage-backed append-only log with multi-algorithm support. The implementation follows the formal model (§IV) directly: the `Log<S: Storage>` type maintains per-algorithm frontier stacks, epoch vectors, and the sealed node store. The `Storage` trait abstracts persistence, admitting in-memory, file-backed, and database backends. The `Hasher` trait abstracts hash algorithms; callers inject implementations at runtime. The crate has zero runtime dependencies.

Table 4: Test suite composition.

Category	Tests	Coverage
Unit tests	51	Append semantics, algorithm lifecycle (`add`, `remove`, or `resume`), proof generation, null-fill correctness, cold reconstruction via `from_storage`, manifest extraction
Property-based (proptest)	16	Equational law verification over thousands of randomly generated tree configurations; state machine test exercising arbitrary interleavings of operations
Fault injection	7	Adversarial storage corruption adapters: bit-flip, truncation, hash drop; includes 2 property-based tests
Complexity regression	6	Empirical curve fitting against formal bounds ($\mathcal{O}(\log n)$ proofs, $\mathcal{O}(1)$ amortized append, $\mathcal{O}(\log K)$ addition, $\mathcal{O}(\log n)$ algorithm resumption)

7.1 Test Suite

The implementation is validated by 80 tests across four categories:

7.2 Complexity Regression

The complexity regression suite empirically validates the performance bounds from Table 2. Each test generates trees at exponentially increasing sizes ($n = 2^4$ through 2^{24}), measures operation timing, and fits the empirical curve against the expected complexity class using the `bigoish` library.

The empirical and theoretical results confirm:

- **Inclusion proofs**: $\mathcal{O}(\log n)$. Proof generation time scales logarithmically with tree size across six orders of magnitude (Figure 5).
- **Consistency proofs**: $\mathcal{O}(\log n)$ by construction. The `subproof` algorithm (Definition 13b) decomposes the tree into a logarithmic number of subtree roots, each resolved via a single stored-node lookup.
- **Append (per algorithm)**: $\mathcal{O}(1)$ amortized. The CTO merge produces at most one sealed node per bit position in the tree size's binary representation.

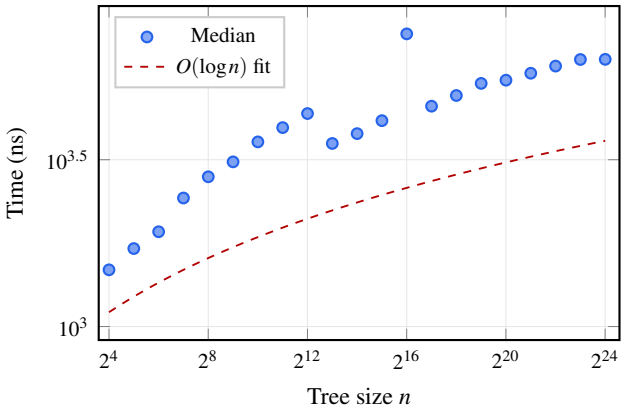

Figure 5: Inclusion proof generation time vs. tree size (log-log). Tree sizes range from 2^4 to 2^{24} ($\sim$16M) leaves; each point is the median of 31 trials (5 warmup) via per-thread CPU time. The dashed line is an $O(\log n)$ fit confirming logarithmic scaling.

Table 5: Fuzz target summary. Corpus size reflects distinct inputs retained after 2×10^6 total executions.

Target	Corpus	Crashes
`verify_inclusion`	134	0
`verify_consistency`	172	0
`rehydrate_proof`	184	0
`proof_mutation`	202	0
Total	**692**	**0**

- **Algorithm addition**: $\mathcal{O}(\text{popcount}(K))$. Null prefix peak computation requires one NullTable entry per set bit in the tree size at the point of addition.
- **Algorithm resumption**: $\mathcal{O}(\log n)$. Frontier reconstruction via `subtree_root` resolution over the binary decomposition of the current tree size, confirmed via curve fitting.

7.3 Fuzz & Fault Injection

The `CorruptingStorage` adapter wraps the storage backend to inject faults (bit-flips, digest truncations, entry omissions). Across all 7 fault-injection tests, mutated proofs are reliably rejected, with zero false positives. Additionally, four `cargo-fuzz` harnesses (Table 5) ran for 2×10^6 executions. The `proof_mutation` target systematically mutates valid proof bits, verifying that all corrupted proofs fail verification. Zero crashes across 692 coverage-guided corpus entries demonstrate that the parser and verification engine handle malformed inputs gracefully.

7.4 Property-Based Testing

We verify structural invariants over randomized tree configurations using 16 proptests. The state machine test executes random sequences of `append`, `add_alg`, `remove_alg`, and `resume_alg` operations, asserting that all invariants (Theorems 1–3 and corollaries) hold after each step. The suite runs 256 cases per test with sizes up to 2^{12} and up to 4 concurrent algorithms.

7.5 Performance Micro-benchmarks

We benchmark SHA-256 inclusion proof generation on a single server CPU thread to evaluate absolute operational latencies. At size $n = 1024$ (height 10), generating an inclusion proof requires a median of 3.58 μs (p25 = 3.54 μs, p75 = 3.62 μs). At $n = 16.7$M leaves (2^{24}, height 24), proof generation requires 6.32 μs (p25 = 6.28 μs, p75 = 6.38 μs), scaling at approximately 196 ns per tree level.

Comparative Overhead Analysis. Under identical storage backend configurations, the database query footprint of the EML matches that of a dedicated single-algorithm Merkle tree (MALT) exactly. EML's `subtree_root` queries issue the exact same database coordinate lookups `get_node(alg_id, left, height)` for active subtrees, and intercept all null intervals in-memory via the local `NullTable`. By routing structural sparsity through in-memory checks, EML avoids duplicate database I/O, inheriting MALT's physical I/O footprint.

8 Related Work

We organize related work by theme, situating the EML against prior and adjacent constructions.

8.1 Foundational & Transparency Log Standards

Merkle introduced authenticated hash trees [12] for signatures. Crosby and Wallach [2] extended them into dynamic *history trees* supporting append-only semantics and $\mathcal{O}(\log n)$ consistency proofs, which form the topological foundation of both Certificate Transparency (CT) and the EML. Pulls and Peeters [16] added forward security in *Balloon* structures. The EML addresses a complementary problem (algorithm lifecycle management) over the same Crosby–Wallach foundation.

RFC 6962 [9] and RFC 9162 [10] standardize CT logs, binding each instance to a single hash algorithm; migrating algorithms requires spinning up a new log identity. Trillian [5] provides a scalable multi-tenant CT backend but inherits this single-algorithm restriction. Sigsum [18] locks down the cryptographic primitives to SHA-256/Ed25519, managing

transitions via time-bounded shard intervals (essentially the "fork the ledger" approach).

The EML retains the single-tree topology of RFC 9162 but permits multiple algorithms to coexist via projections, avoiding both log fragmentation and chimeric weakest-link constructions.

8.2 Algorithm Migration & Sparsity

Git. The SHA-1 to SHA-256 transition in Git [4] requires a bidirectional mapping database between SHA-1 and SHA-256 representations of all objects. This incurs a retroactive translation cost of $\mathcal{O}(N)$ in repository size. The EML's null-fill mechanism avoids this: a new algorithm's projection over historical positions is computed from the local `NullTable` in $\mathcal{O}(\log n)$ on the fly.

IPFS & Sigstore. IPFS Content Identifiers (CID) [14, 15] use self-describing multihashes to enable multiple algorithms within the DHT address space. However, this is per-object; re-hashing content requires full network re-propagation. Sigstore [17] achieves agility through predefined suites selected by policy out-of-band, but does not support concurrent multi-algorithm projections or in-band structural lifecycle management.

Sparse Merkle Trees. SMTs [3, 6, 8, 19] manage sparse keyspaces (e.g., 2^{256} positions) by filling empty leaves with precomputed default hashes. While both SMTs and the EML use deterministic null constants, they differ in the domain of sparsity: SMTs pad a *spatial* keyspace representing potential keys that have no value, compressing proofs with a transmitted bitmap. EML pads a *temporal* axis where an algorithm was inactive, representing real entries that existed before activation. Proof compression in EML uses interval arithmetic over epoch boundaries, requiring no transmitted bitmap (zero per-proof metadata overhead).

9 Conclusion

We have presented the Epoch Merkle Log, a verifiable append-only data structure that resolves the upgrade trilemma for hash transitions. By partitioning algorithm lifecycles into null-padded chronological epochs over a shared RFC 9162 topology, the EML achieves $\mathcal{O}(\log n)$ algorithm addition without retroactive computation, strictly algorithm-independent verification, and zero per-proof-metadata compression via deterministic elision. The formal model (13 definitions, 5 theorems in Lean 4) reduces correctness to standard RFC 9162 verification, and the Rust implementation validates these properties empirically under extensive property-based, fault-injection, and fuzz testing.

Ultimately, EML transforms the complex operational challenge of coordinating hash function transitions across distributed systems into a locally managed, formally verified structural operation. A log operator can register a post-quantum hash algorithm in $\mathcal{O}(\log n)$ time, while standard RFC 9162 verifiers verify proofs without modification, preserving temporal fidelity and preventing ledger fragmentation.

9.1 Future Work

- **Post-quantum integration.** Pairing EML with Merkle Tree Certificates [7] would unify epoch hash management with amortized post-quantum signature costs.
- **P2P replication.** Leveraging EML consistency proofs to build gossip-based log replication protocols at the application layer.
- **Signature agility.** Extending the epoch lifecycle model to asymmetric key rotation cleanly at the transport layer, preserving the underlying tree algebra.

Ethical Considerations

This work presents a defensive cryptographic data structure for hash algorithm lifecycle management in append-only logs. It does not involve human subjects, personal data collection, or interaction with production networks, and required no Institutional Review Board (IRB) approval. We identify no dual-use concerns or negative societal impacts; the Epoch Merkle Log is designed to lower the systemic risks of hash function deprecation in public audit logs. All evaluation was conducted using synthetic test data.

Open Science

We provide the following artifacts for evaluation. All materials are included in the supplementary archive accompanying this submission.

Source code. The complete Rust implementation of the EML, including the `Log<S: Storage>` state machine, the `Hasher` and `Storage` traits, proof generation and verification functions, and the elision/rehydration engine. The crate has zero runtime dependencies; all hash algorithm implementations are injected by the caller via the `Hasher` trait.

Test suite. 80 tests across four categories:

- 51 unit tests covering append semantics, algorithm lifecycle, proof generation, null-fill correctness, and cold reconstruction.
- 16 property-based tests (proptest) verifying equational laws over randomized tree configurations, including a comprehensive state machine test.
- 7 fault injection tests using adversarial storage adapters.
- 6 complexity regression tests validating asymptotic bounds via curve fitting.

All tests are reproducible via `cargo test` (unit, proptest, fault injection) and `cargo test --release --test complexity` (complexity regression).

Fuzz harnesses. 4 `cargo-fuzz` targets exercising adversarial inputs against proof verification and elision surfaces. Reproduction: `cargo +nightly fuzz run <target>`.

Formal model and Lean 4 proof. The formal model (13 definitions, 5 theorems) and its complete machine-checked verification (establishing structural equivalence, algorithm isolation, and temporal binding) are located in `proofs/lean/` and `docs/models/`. The Lean 4 proof consists of approximately 1400 lines of proof code checking 30+ theorems with zero `sorry` axioms.

Reproduction instructions. A `README` in the supplementary archive documents the exact commands to build, test, and reproduce all results reported in §VII. The development environment is reproducible via a Nix shell specification (`shell.nix`).

References

[1] Daniel J. Bernstein. Boring crypto. Invited talk at Yale University, 2015.

[2] Scott A. Crosby and Dan S. Wallach. Efficient data structures for tamper-evident logging. In *Proceedings of the 18th USENIX Security Symposium*, pages 317–334, Montreal, Canada, August 2009. USENIX Association.

[3] Diem Developers. Jellyfish Merkle tree. Technical report, Diem Association, 2021.

[4] Git Project. Hash function transition. Technical Documentation.

[5] Google LLC. Trillian: A scalable, multi-tenant implementation of verifiable data structures. GitHub Repository, 2016.

[6] Faraz Haider. Compact sparse Merkle trees, 2018.

[7] IETF PLANTS Working Group. Merkle tree certificates. Internet-Draft draft-ietf-plants-merkle-tree-certs-03, Internet Engineering Task Force.

[8] Ben Laurie and Emilia Kasper. Revocation transparency. Technical report, 2012.

[9] Ben Laurie, Adam Langley, and Emilia Kasper. Certificate transparency. RFC 6962, Internet Engineering Task Force, June 2013.

[10] Ben Laurie, Eran Messeri, and Rob Stradling. Certificate transparency version 2.0. RFC 9162, Internet Engineering Task Force, December 2021.

[11] Chujiao Ma, Luis Colón, Joe Dera, Bahman Rashidi, and Vaibhav Garg. CARAF: Crypto agility risk assessment framework. *Journal of Cybersecurity*, 7(1):tyab014, 2021.

[12] Ralph C. Merkle. *Secrecy, Authentication, and Public Key Systems*. PhD thesis, Stanford University, June 1979.

[13] Ralph C. Merkle. A certified digital signature. In *Advances in Cryptology — CRYPTO '89 Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer, 1990.

[14] Protocol Labs. Content identifiers (CIDs). IPFS Documentation.

[15] Protocol Labs. Multiformats: Self-describing values for future-proofing. Specification.

[16] Tobias Pulls and Roel Peeters. Balloon: A forward-secure append-only persistent authenticated data structure. In *Computer Security – ESORICS 2015*, volume 9327 of *Lecture Notes in Computer Science*, pages 622–641. Springer, 2015.

[17] Andrea Schirone. Building cryptographic agility into Sigstore. Trail of Bits Blog, 2026.

[18] Sigsum Project. Sigsum logging design v0. Design Document, 2021.

[19] Tari Development Community. RFC-0141: Sparse Merkle trees. Technical report, Tari, 2018.