DOI: https://doi.org/10.15276/hait. 09.2026.32
UDC 004.021:004.422.63:004.056.55

Fractal principles for constructing second-order cryptographic accumulators

Artem A. Buchikhin[1)]
ORCID: https://orcid.org/0009-0006-9562-1273; artembuchihin@gmail.com
Viktor O. Boltenkov[1)]
ORCID: https://orcid.org/0000-0003-3366-974X; vaboltenkov@gmail.com. Scopus Author ID: 57203623617
[1)] Odesa Polytechnic National University, 1, Shevchenko Ave. Odesa, 65044, Ukraine

ABSTRACT

Cryptographic accumulators are a core component of blockchain verification systems, including zero-knowledge bridges, where proof size directly impacts on-chain cost and prover performance. Existing accumulator designs, such as Merkle Mountain Ranges and their variants, provide strong asymptotic guarantees but do not adequately exploit the temporally biased access patterns typical of real-world verification workloads. **The aim of work** is to present and analyze a new cryptographic accumulator data structure that minimizes the expected proof size under realistic, time-varying verification workloads.In this work, a new class of accumulator data structures, the Fractal Merkle Tree and Fractal Loom Tree, is introduced and analyzed with the goal of minimizing expected proof size under such workloads. Proof cost is formalized in terms of cryptographic hash operations and input word counts, reflecting both on-chain verification expenses and in-circuit prover workload. The Fractal Loom Tree construction employs multi-scale pointer selection, yielding verification costs that grow logarithmically with the temporal distance from the most recent state. The parameter space of Fractal Loom Tree is characterized, showing how radix, tree depth, and pointer allocation shape the time-dependent distribution of proof sizes. An anchor caching optimization is presented, enabling reuse of persistent intermediate states and further reducing worst-case verification cost. Directions for future research are proposed to extend the construction to alternative commitment primitives, adaptive parameter selection, and more general graph-based structures. The main result of this work is the development of a cryptographic accumulator that achieves substantially smaller proofs than existing designs under realistic, temporally biased blockchain verification workloads.

Keywords: Cryptographic accumulators; Merkle trees; zero-knowledge proofs; zk-bridges; blockchain verification, proof size optimization; commitment data structures; verifiable data structures

***For citation*: Buchikhin A. A., Boltenkov V. O. "Fractal principles for constructing second-order cryptographic accumulators". *Herald of Advanced of Information Technology*. 2026; Vol.9 No.4: 502–520. DOI: https://doi.org/10.15276/hait.09.2026.32**

INTRODUCTION

Blockchain interoperability, though relatively young, has become one of the fastest-evolving areas of Web3 research. This pace is justified by the scale of the underlying problem. Blockchains provide strong internal security guarantees, yet lack native mechanisms for verifying information originating from independent ledgers governed by different consensus rules and state transitions [1]. A chain can only validate what it computes itself; statements about external state are unverifiable unless additional cryptographic machinery is introduced to prevent forgery [2]. Over the past few years, zero-knowledge bridges have emerged as the dominant approach to this problem, offering both scalability and strong correctness guarantees [3]. By allowing one chain to verify succinct proofs of execution on another, zk-bridges eliminate trust in relayers and avoid the latency and failure modes of optimistic designs [4].

© Buchikhin A., Boltenkov. V., 2026

However, despite their conceptual elegance, zk-bridges remain expensive to operate. A substantial fraction of their cost is not attributable to cryptographic primitives themselves, but to the verification of long membership proofs over growing block histories against periodically anchored commitments [5]. These proofs expand with chain age, inflate zero-knowledge circuits, and directly translate into higher gas usage, longer proving times, and increased fees for users.

As zk-bridges transition from experimental infrastructure to production-critical systems, this cost structure becomes a limiting factor. Without architectural changes to how blockchain history is committed and verified, improvements at the level of hash functions or circuit optimizations alone cannot prevent proof costs from scaling over time. Addressing this structural inefficiency early is essential, before current accumulator designs become entrenched as long-term standards.

The purpose of the paper is to introduce and analyze a new cryptographic accumulator data structure that minimizes expected proof size

This is an open access article under the CC BY license (https://creativecommons.org/licenses/by/4.0/)

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

under realistic, temporally biased verification workloads.

The objectives of the study are as follows:

– formalize the notion of proof cost in terms of cryptographic hash operations for zk-based bridge verification;

– analyze the limitations of existing accumulator data structures under temporally biased access patterns;

– present and formally define second-order cryptographic accumulators, as well as Fractal Merkle Tree (FMT) and Fractal Loom Tree (FLT) data structures;

– characterize the tweakable parameters and observable properties of FLT and their impact on proof size and its time-dependent distribution;

– introduce anchor caching as an optimization technique and analyze its interaction with FLT;

– provide and evaluate a concrete parameterization relevant to large-scale blockchain histories.

The article is structured as follows. Section 2 reviews related work on accumulator data structures. Section 3 formalizes the problem setting and discusses implementation constraints in zk-bridge architectures. Section 4 defines and analyzes fundamental properties of the Fractal Merkle Tree (FMT) and Fractal Loom Tree (FLT) data structures. Section 5 formalizes the parameter space of FLT and examines how its configuration affects proof size over time, along with other structural properties. Section 6 presents anchor caching and other optimization techniques. Section 7 evaluates the proposed approach through a concrete configuration and workload model. Section 8 concludes the paper and outlines directions for future research.

RELATED WORKS

Accumulator data structures enable iterative commitment to an append-only sequence while supporting succinct membership proofs. They form the cryptographic backbone of light clients, bridges, and rollup verification systems, where both update efficiency and proof verification cost are critical. Existing constructions differ significantly in how proof size scales with history length, how efficiently updates can be performed, and how suitable the structure is for verification inside smart contracts and zero-knowledge circuits.

Classical Merkle trees represent the most widely used hash-based accumulator [6]. They provide logarithmic-size membership proofs and simple verification logic. However, their structure treats all historical elements symmetrically, causing proof size and verification cost to grow monotonically with the total number of committed elements. Incremental variants mitigate rebalancing costs but do not change this fundamental property [7]. As a result, Merkle trees lack any notion of temporal locality: proving membership of a recent element is asymptotically as expensive as proving membership of the first element ever committed. This makes them poorly suited for long-lived systems such as blockchain bridges, where the majority of queries concern recent blocks.

Merkle Patricia Tries (MPTs), used in Ethereum for state commitments, extend Merkle trees to support dynamic key–value mappings [8]. While expressive, their proof sizes are highly variable, depend on path structure, and involve complex encodings. This results in unpredictable verification cost and poor compatibility with constraint-based verification environments. MPTs are therefore optimized for mutable state representation rather than efficient append-only history accumulation.

Algebraic and polynomial commitment schemes, including KZG commitments, IPAs, and RSA accumulators, can provide proofs with asymptotically sublinear or constant size [9], [10], [11]. However, achieving this property requires more complex cryptographic primitives, higher prover computational cost, more complex verification procedures, and, in some cases, trusted setup assumptions. In zero-knowledge circuits, verifying such commitments typically incurs orders of magnitude more constraints than evaluating cryptographic hash functions. Consequently, despite their theoretical appeal, algebraic accumulators remain impractical for high-throughput zk-bridge verification, where verification cost is dominated by arithmetic constraint complexity.

Merkle Mountain Ranges (MMRs) were introduced to support efficient append-only accumulation without tree rebalancing [12]. They maintain a set of perfect binary subtrees (“mountains”) whose roots are aggregated into a single commitment. MMRs offer logarithmic-size proofs and constant-time append operations and are widely deployed in bridge designs such as BEEFY and Snowbridge. Nevertheless, their membership proof size grows with the total number of elements, and verification requires hashing across an increasing number of peaks. As the structure ages, verification cost steadily increases even for recent elements, and partial proof reuse or caching is difficult due to the global dependence on peak aggregation.

Several variants generalize MMRs to higher branching factors, such as k-ary MMRs (KMMRs), reducing tree depth and moderately decreasing proof size [13]. However, these constructions preserve the same asymptotic behavior: proof cost remains a function of total history size and exhibits no sensitivity to the recency of the proven element.

Merkle Mountain Belts (MMBs) were recently proposed to address this limitation by introducing temporal locality into the proof structure [14]. Instead of aggregating all peaks into a single root, MMBs organize them into a sequence of "bags" forming a belt, such that recent mountains are closer to the commitment root while older ones are gradually pushed deeper into the structure. This yields membership proofs whose size depends primarily on how recent the queried element is, rather than on the total number of accumulated elements. While effective for workloads dominated by recent queries, MMBs introduce significant structural complexity, require overwriting and re-aggregation of peaks, limit opportunities for anchor caching, and fix a particular trade-off between update cost, recency window, and historical depth. The design space of possible cost–time trade-offs remains largely implicit and difficult to tune.

The analysis above suggests that the core limitation of existing accumulator designs is not merely proof size, but the lack of an explicit abstraction for time. Current structures encode temporal order implicitly through hashing order or graph topology, coupling ancestry, data layout, and commitment evolution into a single mechanism. This prevents stable verification depth, reuse of intermediate structure, and controlled amortization. What is required instead is an accumulator in which temporal relationships are governed by an independent, deterministic rule, while the data associated with each moment in time remains unconstrained. This work introduces Fractal Merkle Trees (FMT) and Fractal Loom Trees (FLT) as a family of second-order cryptographic accumulator data structures that make this trade-off explicit through configurable structural parameters, enabling systematic optimization of proof cost for zk-bridge verification under varying workloads and time horizons.

ANALYSIS OF REQUIREMENTS FOR THE TARGET DATA STRUCTURE

Operational structure of zk-bridges. A zero-knowledge proof is a cryptographic protocol that allows one party (the prover) to convince another party (the verifier) of the correctness of a particular computation without requiring the verifier to re-execute the computation and without disclosing intermediate data. Zero-knowledge bridges use such proofs to enable verification of events from a source blockchain on a destination blockchain without relying on trusted intermediaries [15].

In deployed systems, this verification process is organized into three logically distinct phases. First, an anchor synchronization phase periodically publishes to the destination chain a cryptographic commitment to the source chain's state of history [16]. Second, for each client request, a block-to-anchor proof demonstrates that a particular block belongs to the history committed by the anchor [17]. Third, an in-block proof establishes that a specific transaction, receipt, or state transition is included in that block [18]. Only the latter two phases are executed per client transaction, while anchor synchronization is amortized over many users.

An anchor may take multiple forms depending on the bridge design: a commitment to block headers, to a sequence of state roots, or to an accumulator summarizing the entire chain prefix [19]. Its validity may be guaranteed by a consensus proof, validator signatures, fraud-proof windows, or a separate zk-proof [20]. The construction and verification of anchors is therefore highly system-specific and orthogonal to the present work. We only assume that anchors are cryptographically binding commitments to an append-only block sequence and that their verification enforces standard accumulator requirements such as collision resistance, deterministic updates, and efficient membership proofs.

Where optimization is possible. The in-block proof component is determined by the internal data model of the source blockchain, typically a Merkle Patricia Trie or an equivalent authenticated structure. Altering its cost would require protocol-level changes and ecosystem-wide migration, which in practice unfolds over multi-year timescales. As a result, this part of the verification pipeline is effectively fixed.

In contrast, the block-to-anchor proof depends entirely on the accumulator data structure chosen to represent block history. This proof establishes that a given block hash appears at some position in the committed sequence, a fact guaranteed by the security of the underlying commitment primitive. Consequently, block-to-anchor verification is the only layer where structural optimization is both technically feasible and immediately deployable without modifying the source blockchain.

Since both the block-to-anchor proof and the in-block proof must be verified for each client

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

transaction, even a relatively small reduction in their individual cost is accumulated across the entire lifetime of a bridge. For systems processing hundreds of thousands or millions of cross-chain interactions, this accumulated cost directly affects infrastructure requirements, prover throughput, and the available capacity for on-chain verification.

Current standard and verification modalities. Merkle Mountain Ranges and their derivatives are currently the de facto standard accumulator used in zk-bridges for block-to-anchor proofs, due to their append-only nature and efficient updates. Variants such as KMMR and Merkle Mountain Belts modify the update or bagging procedure but retain the same fundamental proof structure.

Architecturally, block-to-anchor proofs are verified either directly on-chain by a smart contract or inside a zero-knowledge circuit together with the in-block proof. These two approaches have different deployment and trust assumptions, but the verification procedure remains essentially the same. The verifier recomputes the cryptographic hashes along the inclusion path and compares the resulting value with the anchor. In a zk-circuit, these hash computations require a large number of arithmetic constraints. During on-chain verification, the same computations contribute directly to gas consumption. In both cases, the total cost is therefore proportional to the number of hash invocations required by the inclusion proof.

This yields a natural and implementation-independent cost model in which the efficiency of an accumulator is measured primarily by the expected hash count per block-to-anchor proof.

Temporal locality of verification workload. Cross-chain activity is not uniformly distributed over historical blocks. Economic processes impose strong temporal locality: recent events are far more likely to be referenced than distant ones. Arbitrage, liquidations, oracle updates, rollup withdrawals, governance execution, and bridge-mediated contract calls all predominantly target blocks produced minutes or hours earlier, while the probability of referencing older blocks decays rapidly over time. Although public datasets for zk-bridge usage are scarce, this pattern mirrors transaction lifecycles observed in payment systems, exchanges, and settlement networks, where value relevance follows a heavy-tailed, approximately exponential decay with respect to time.

This implies that the cost of verifying blocks from the most recent hours and days dominates real-world bridge workloads, even though the accumulator must commit to millions of blocks overall. An optimal data structure should therefore minimize proof size not only asymptotically, but specifically for recent history, while preserving succinct commitments to the full chain.

Optimization target. The objective of this work is to design accumulator data structures that replace MMR-style constructions in zk-bridges and minimize the number of hash operations required for block-to-anchor proofs, particularly for recent blocks, under the constraints imposed by anchor-based synchronization. The following sections introduce such structures and formalize how their parameters control proof cost over time.

SECOND-ORDER CRYPTOGRAPHIC ACCUMULATORS

This section introduces the distinction between first-order and second-order cryptographic accumulation and defines the second-order accumulator model considered in this work. The Fractal Merkle Tree (FMT) is subsequently introduced as a concrete realization of this model, along with the underlying node model, the pointer-based temporal layer, and the abstraction on which the subsequent Fractal Loom Tree (FLT) construction is built.

First- and Second-Order Cryptographic Accumulators. In a first-order cryptographic accumulator, the elements being accumulated are externally defined objects. Their internal representation and cryptographic structure are determined independently of the accumulator, while the accumulator combines their commitments into a larger authenticated structure. A conventional Merkle tree represents this approach, where externally defined data elements are committed and then combined according to the selected tree construction.

A second-order cryptographic accumulator is an accumulator in which the accumulated elements are themselves cryptographically structured objects whose construction or commitment depends on the accumulator configuration or state. One of its core properties is that structural decisions made at the node level of abstraction and above directly propagate to the internal structure of the accumulated nodes. In other words, the accumulator not only combines cryptographic commitments of externally defined elements, but also participates in determining the structure of the elements that are accumulated.

This property allows cryptographically structured elements to be recursively incorporated into subsequent authenticated structures. An

authenticated structure constructed for one element can therefore become a component of another authenticated structure without requiring duplication of the underlying data.

Fractal Merkle Tree. The Fractal Merkle Tree (FMT) is a second-order cryptographic accumulator in which each accumulated node consists of two components: a temporal component and a content component.

The temporal component is a deterministic set of pointers to earlier nodes generated by a universal rule at creation time. The content component is a cryptographic commitment to protocol-specific data, such as block headers, batch roots, or upgrade instructions. While the structure itself is agnostic of the underlying commitment primitive, nodes may be constructed by a hash or by an internal cryptographic tree of arbitrary form.

The temporal components induce a directed acyclic pointer structure that defines reachability and ancestry. This structure is independent of the committed data and is determined by the construction of the accumulator. The content component is attached to this structure and does not determine the temporal relations between nodes.

An FMT therefore defines the temporal geometry of the accumulated history separately from the committed content. Unlike a conventional Merkle tree, where the placement of accumulated elements determines the resulting tree structure, the FMT temporal structure is established by the rule used to construct its nodes. This provides the basis for the traversal properties considered in the following sections.

The term Fractal Merkle Tree is used in this work to denote this specific construction and its associated definition. It is not intended to assert identity with other structures that may have previously used the same name.

Pointer Tree: Temporal Identity of a Node. A pointer tree is an acyclic directed hash structure rooted at N_i whose leaves are references to earlier nodes.

Formally, each node selects a finite admissible subset of predecessors:

$$P(i) = \{ j \mid 1 \le j < i,\ j \text{ is permitted} \}. \quad (1)$$

The set of permitted indices is determined by the pointer tree definition, which specifies the range and pattern of allowable backward references.

Importantly, the definition of which references are allowed does not prescribe which ones are chosen. Automated pointer-selection functions may be applied, but they are optional and external. The pointer tree therefore represents a space of permissible temporal edges (slots), not a specific rule for deterministically generating them. This independence ensures that the reachability structure of the cryptographic geometry of time does not depend on the payload or on the total length of the history. It also allows different applications to instantiate nodes either by deterministic rules or by deliberative processes (e.g., governance-driven versioning), as long as each new node's pointers fall within the permitted set.

Content Attachments: State Binding and Action Binding. While the pointer tree governs the temporal structure, the payload bound to a node is entirely external. The FMT imposes only a minimal requirement: the payload must be representable as a hash that can be fused with the pointer structure. Two classes of payloads are of interest.

A state-binding node commits to a digest summarizing the state (any type of data or a delta of update that occurred since the previous node). To ensure that this state change is inseparably tied to the node's temporal identity, the payload is fused with the current pointer-tree root.

Let $state_i$ denote the external state at time i, and let $ptrRoot(N_i)$ be the canonical hash representing the pointer tree of N_i. The canonical root of the node is defined as:

$$root(N_i) = H\big(state_i \ \| \ ptrRoot(N_i)\big), \quad (2)$$

which is then referenced by future nodes. This "forward fusion" binds every state or state update to its temporal position.

In action-binding configurations, a node's payload describes an externally interpretable transition, such as a reversible step, emergency control primitive, or policy-authorized operation. To guarantee that the action is bound to the exact historical context from which it is permitted, the payload is fused with the previous node root. This "backward fusion" embeds the action into the temporal chain, ensuring that actions are cryptographically tied to the state from which they originate. Action semantics remain outside the FMT; the accumulator merely authenticates their temporal placement.

The remainder of this work focuses on state-binding nodes, because they form the analytical core of continuity structures used in zk-bridges and other state-commitment mechanisms. Action-binding nodes rely on the same pointer-tree framework but do not affect the topology or proof-theoretic properties central to the latency analysis. For the state-binding case considered here, we begin with

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

defining the simplest case where payload is a single state (block) hash fused with its pointer-tree root.

Formal definition of a node (state-binding case). A node N_i in an FMT consists of two binding layers:

– a pointer-tree structure whose leaves are references to earlier node roots;

– a state-binding hash representing the state associated with time *i*.

These two components are fused into a single canonical commitment, the node root, which becomes the anchor used by all future nodes, as shown in Fig. 1.

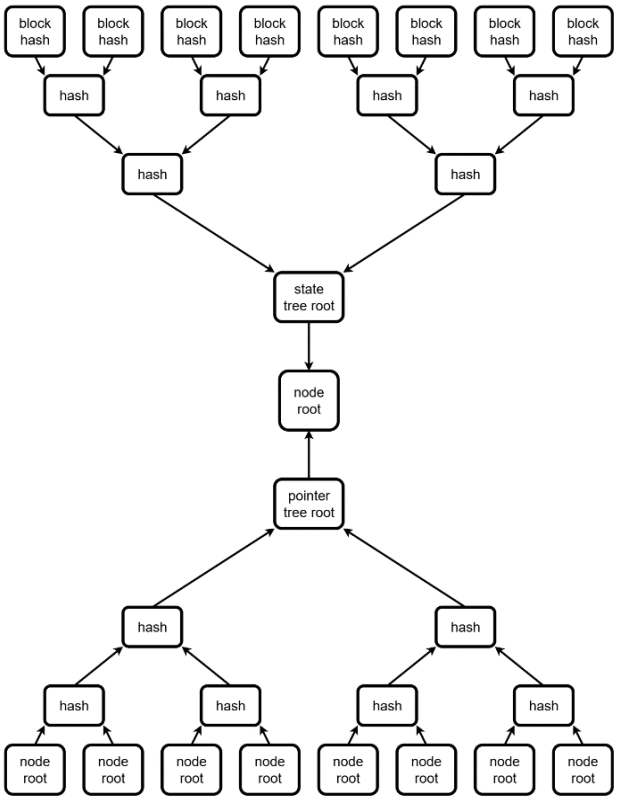


Fig. 1. **Structure of an FMT node**
Source: **compiled by the authors**

For pointer tree, let *A(i)* be the set of admissible backward references ("allowed predecessors"). A particular instantiation of node N_i selects a finite nonempty subset $P(i) \subseteq A(i)$.

The node constructs a Merkle-style tree (pointer tree) whose leaves are precisely the roots of referenced earlier nodes:

$$L_i = \left(root\left(N_{j_1}\right), \ldots, root\left(N_{j_k}\right)\right) \quad for\ P(i) = \{j_1, \ldots, j_k\}. \tag{3}$$

The pointer-tree commitment is:

$$ptrRoot\left(N_i\right) = Merkle\left(L_i\right). \tag{4}$$

This defines the temporal identity of the node.

For state tree, let $state_i$ be the hash representing the external state (chunk of data, delta, batch-root, or any summary) associated with time of inclusion *i*. This hash is not prescribed by the pointer tree and may represent any state commitment of the application.

To bind the state to its temporal identity, the state hash is fused with the pointer-tree root (2). This fused root is the only value of N_i used in future pointer trees.

Thus, all components of node definition for our case are complete, and state-binding node in an FMT is the tuple:

$$N_i = (P(i),\ state_i,\ ptrRoot\left(N_i\right),\ root\left(N_i\right)) \tag{5}$$

where *P(i)* is the selected predecessor set; $ptrRoot(N_i)$ commits to the pointer tree built from these predecessors; $state_i$ is an arbitrary external state commitment; $root(N_i)$ is the canonical commitment used by all future nodes.

This construction fully separates temporal geometry (pointer tree) and payload semantics (state), while ensuring that both are cryptographically bound at each point in time.

Node Organization. An FMT can have various node organization structures, but in our case, we will narrow the scope down to a linear and append-only one. Here FMT is defined as a set of nodes

$$\mathrm{N} = \{N_1, N_2, \ldots\}, \tag{6}$$

with a total order $N_0 \prec N_1 \prec \ldots$, that represents the progression of time.

The set itself is unordered at the data-structural level; the temporal ordering is an external constraint imposed on node creation. This ordering is append-only: a new node N_i may be created only after all nodes $N_0, \ldots, N_{i-1}$ already exist, and, crucially, no node may be inserted between existing indices. This linear organization ensures that each node possesses a unique temporal position, which is indispensable for deterministic verification. It also ensures that the pointer-tree roots of earlier nodes are immutable: once a node's root becomes part of history, it can only appear as a leaf in the pointer trees of later nodes, never be modified or removed.

While the timeline is linear, the temporal relations encoded by nodes are not. Each node N_i links to a deterministic (or externally chosen) set of

ancestors specified by its pointer tree. This allows the history to be navigated through multiple structured backward edges rather than only through the immediate predecessor. This combination of linear time and non-linear ancestry is the structural distinction that separates the FMT from classical Merkle trees and enables its graph-theoretic behavior.

Determinism of ancestry is essential: for any chosen pointer policy, two independent observers must derive the same set of allowable ancestors for a node at index i. This preserves the ability of a verifier to reconstruct and validate the temporal structure from publicly known rules and commitments alone, without additional metadata. Thus, the FMT may be viewed as a linearly ordered set of cryptographically bound DAG (Decentralized Acyclic Graph) vertices, each of which introduces a structured, verifiable pattern of edges into the past.

The Loom Mechanism. The pointer tree defines where a node may point; the Loom defines how those pointers are instantiated at each step of the timeline. The Loom is a general generative mechanism that accepts a pointer-selection function and applies it consistently to every newly appended node.

Let the FMT contain a linearly ordered set of nodes $\mathcal{N}$, and let $i \in \mathbb{N}$ denote the index of the node currently being appended. Each node N_i must select a finite set of predecessors

$$P(i) \subseteq \{1, 2, \ldots, i-1\} \tag{7}$$

to populate the leaves of its pointer tree. The Loom determines this set through a pluggable function, hence being a deterministic operator:

$$\mathrm{L}: \mathrm{N} \times \Phi \to 2^{\mathrm{N}}, \tag{8}$$

where Φ is the class of admissible pointer-selection functions. Given a function $f \in \Phi$, the Loom produces for each index i the predecessor set

$$P(i) = \mathrm{L}(i, f), \tag{9}$$

defined by the rule:

$$P(i) = \bigcup_{k=1}^{m_i} \{ f(i,k) \cap \{1, 2, \ldots, \mathrm{i}-1\} \}. \tag{10}$$

Here:

– *f(i,k)* is the k-th predecessor proposed by the pointer-selection function *f* (or k-th slot in pointer tree);

– m_i is the number of predecessors requested by *f* at index *i* (finite for all *i*);

– the intersection ensures that only backward edges remain valid.

In other words, the pointer-selection function *f* produces a sequence of candidate predecessor indexes, and the Loom filters, validates, and assembles them into the actual pointer set. The notation above emphasizes that the pointer tree is built as the aggregation of individually selected backward references, each generated by evaluating *f* at *(i,k)*.

The Loom enforces the minimal structural constraints required for the FMT to remain a valid, verifiable Merkle-DAG:

1. Backward-only constraint $f(i,k) < i$ ensures acyclicity.

2. Deterministic reconstruction constraint enforces the verifier to reconstruct *P(i)* exactly from *f* and public parameters; no hidden metadata is allowed.

3. Locality constraint, that makes $P(i) = \mathcal{L}(i,f)$ dependent only on *i* and global parameters of *f*. It cannot depend on future nodes.

4. Finite fan-out constraint, where for every *i*, $m_i < \infty$, so the pointer tree is Merkle-committable.

5. Reachability constraint, as the function *f* must guarantee that all earlier nodes remain reachable from N_i via a bounded number of pointer hops; the bound depends on the family of functions chosen in later sections.

These constraints ensure that the FMT remains a parameterized generative mechanism, not tied to any specific pointer geometry.

The Loom does not predetermine the topology of the FMT. It merely supplies the rule by which each new node in the linear timeline fills its pointer tree with admissible backward references. Different functions *f(i,k)* plugged into the Loom produce different temporal geometries but all of these are instantiations, not part of the Loom's definition. The specific pointer-selection functions that yield the Fractal Loom Trees (FLT), the Full-Reachability FMT (FR-FMT), and other traversal structures will be introduced next.

Topological Cases of FMT. The introduced Loom mechanism allows us to define a wide class of Fractal Merkle Trees (FMTs) by specifying a deterministic rule for predecessor selection at each node insertion. Under this abstraction, different FMT cases are not distinguished by their cryptographic primitives or node composition, but purely by the topology of temporal dependencies they induce. Despite this flexibility, all valid FMT topologies developed in this work are constrained by a common set of invariants.

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

First, the structure must be sequentially generative: nodes are appended one by one, and the Loom rule must be computable solely from the index of the new node and previously constructed nodes. No lookahead, rebalancing, or global restructuring is permitted. This ensures that FMT construction remains streamable and compatible with incremental data aggregation in real-time systems.

Second, every topology must provide strong reachability continuity across time. Specifically, the same reachability rule must apply for any appended node, independent of its position in the sequence, guaranteeing uniform traversal properties across the entire history. As a minimal requirement, this implies that each node must reference its immediate predecessor, ensuring that trivial reachability (i.e., stepping back one position at a time) is always possible, even in the absence of higher-order links.

Third, an FMT topology must preserve eventual full historical reachability: from any node, every previous node must be reachable via a finite sequence of Loom-defined hops. This condition ensures that all FMT instances represent not merely partially linked histories, but fully navigable temporal graphs.

Within these constraints, the Loom function defines distinct topological cases of FMT, each characterized by how temporal links are distributed across the history.

Following the traversal complexity rules three such cases exist:

– Linear FMT (LFMT) – each node points only to the previous node (like in blockchains);

– Full-Reachability FMT (FR-FMT) – each node points to all previous nodes;

– Fractal Loom Tree (FLT) – each node is weaved into the structure to point to previous nodes in such a way that allows to reach any of them in logarithmic number of steps.

Each one of them explores a different regime of the design space, trading off pointer density, traversal efficiency, and structural complexity while preserving the same underlying FMT construction semantics. We will directly focus on the last one, since linear case is just a Merkle chain which leads to linear growth of hashings in inclusion proof as more nodes appear, while full-reachability case gets to inflate the pointer tree quick, which also leads to a substantial number of hashes for the same purpose (on the other hand it's good for the cases with limited number of nodes, like versioning).

Fractal Loom Tree (FLT): Binary Loom. The Fractal Loom Tree introduces a logarithmically structured predecessor selection rule, in which each node references a small, structured subset of its predecessors determined by a hierarchical decomposition of its index. In this subsection, we consider the binary case, which forms the conceptual and algorithmic foundation of the general construction.

Unlike LFMT and FR-FMT, where pointer selection is either purely linear or fully dense, the binary FLT employs a sparse, structured selection that guarantees logarithmic historical reachability while preserving sequential, streamable construction.

Let nodes be indexed N_1, N_2, …, N_i with $i \geq 1$. We define three sequences:

$$\begin{aligned} &(s_r)_{r\geq 1}, \\ &(\ell_r)_{r\geq 1}, \\ &(j_r)_{r\geq 1}, \end{aligned} \tag{11}$$

which describe the successive decomposition of the index space associated with node N_i. Initially

$$s_1 = 1, \quad \ell_1 = i-1, \tag{12}$$

so that the initial local interval is

$$[s_1,\ s_1+\ell_1-1] = [1,\ i-1], \tag{13}$$

which means all valid predecessors of N_i. For the recursive step, for each $r \geq 1$, provided $\ell_r > 0$, we define:

1. The largest power of two not exceeding the current slice length:

$$p_r = 2^{\lfloor \log_2(\ell_r) \rfloor}. \tag{14}$$

2. The selected predecessor index:

$$j_r = s_r + p_r - 1. \tag{15}$$

3. The next slice (the suffix strictly after j_r):

$$s_{r+1} = j_r + 1, \quad \ell_{r+1} = i - s_{r+1}. \tag{16}$$

4. Process termination condition at the smallest r such that $\ell_{r+1} = 0$.

Let $m_i = r$ denote the number of iterations executed by the binary decomposition algorithm for node N_i. For each pointer slot $k=1, \ldots, m_i$, let j_k denote the predecessor index j_r obtained at step r of the algorithm, with the identification $r \equiv k$.

Then the resulting binary pointer-selection function is defined as:

$$f_{bin}(i,k)=\begin{cases} j_k, & 1\le k\le m_i, \\ undefined, & k>m_i. \end{cases} \quad (17)$$

The Loom aggregates these into the predecessor set:

$$P_{bin}(i)=\bigcup_{k=1}^{m_i}\{f_{bin}(i,k)\}. \quad (18)$$

The rest of the algorithm remains trivial. Thus, each node N_i references exactly the set of predecessors determined by the binary decomposition of its index range *[1, i−1]*.

This construction greedily decomposes the interval of predecessors using powers of two, selecting exactly one pointer per level of binary magnitude (Fig. 2).

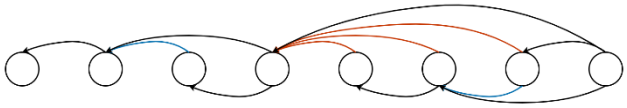


Fig. 2. **DAG-representation of Binary FLT**
Source: **compiled by the authors**

The black arrows here show the stable path to all previous nodes, which appears upon attaching a node, whose index is a power of 2. The orange and blue arrows represent the additional pointers at respective power levels required at the moment of weaving of each separate node into the already accumulated structure.

As a result, the number of pointers per node grows as $O(\log_2(i))$. Any node N_i can reach any preceding node N_j via at most $O(\log_2(i))$ hops. The structure remains sequential and streamable: pointer selection depends only on the current index i and previously materialized node roots.

In contrast to the linear structure of LFMT and the dense expansion of FR-FMT, the binary FLT introduces a hierarchical temporal topology, where predecessors are sampled at exponentially increasing distances. This binary formulation serves as the conceptual baseline for the generalized FLT construction, in which the powers of two are later replaced by higher powers of a radix $b = 2^a$, allowing finer control over the trade-off between pointer density and traversal complexity.

The Fractal Loom Tree family. The Fractal Loom Tree (FLT) constitutes the primary contribution of this work. It introduces a tunable, hierarchical temporal topology over an append-only sequence of nodes, enabling logarithmic historical traversal while remaining fully sequential and streamable. Unlike LFMT and FR-FMT, which represent fixed extremes of pointer sparsity and pointer density, FLT defines a parameterized design space controlled by a single architectural knob: the radix $b = 2^a$, $a \in \mathbb{N}$.

This radix determines the scale hierarchy used by the Loom mechanism and thereby shapes the structure of temporal reachability. Small radix values (e.g., $b = 2$) produce sparser topologies with deeper traversal, while larger radices (e.g., $b = \{4, 8, 16\}$) produce flatter, more pointer-rich structures with shorter traversal. All variants preserve the essential constraints of the FMT framework: sequential construction, uniform reachability rules, and finite-hop access to all prior nodes.

To motivate the general construction, we first consider the binary variant $b = 2$. In this setting, the index interval *[1, i−1]* is decomposed into slices whose lengths shrink by powers of two. At each scale 2^n, the Loom selects the largest power of two that fits into the remaining suffix, effectively partitioning the interval into 2^{n1}, 2^{n2}, … segments.

The binary formulation is simple because each level has exactly one nonzero multiple $p \cdot 2^n$, $p \in \{1\}$. Higher multiples do not exist, so no choice is required. This “one pointer per scale” behavior makes the binary case an instructive foundation, but also exposes its limitation: its coverage of the interval is minimal. To reduce hop depth, the structure must leverage wider scales.

For radix $b=2^a$ where $b > 2$, each scale level ℓ_b admits a set of nonzero multiples:

$$\{pb_n : 1\le p\le b-1,\ pb_n\le \ell_b\}. \quad (19)$$

Restricting to a single multiple per level would dramatically underutilize the available scale resolution and produce a structure no flatter than the binary case. To exploit the expressive power of higher radix values, FLT adopts the principle of full-level coverage: At each level b_n, the Loom includes all multiples pb_n that fit into the current slice, stopping only once a multiple exceeds the remaining length.

This design accomplishes three objectives simultaneously. Multiple pointers at each scale collapse large portions of the index interval in a single step, reducing hop depth below the binary bound. The entire suffix is partitioned into exponentially scaled blocks, ensuring reachability of every ancestor. Because all predecessor indices are deterministic functions of i, the construction remains streamable, requiring no global restructuring.

In short, the binary case selects a single anchor per scale, while the general case selects all viable anchors, compressing the interval more aggressively.

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

The resulting pointer set per node grows to at most $(b-1)\lfloor \log_b(i) \rfloor$, but pointer-tree depth remains hyperlogarithmic relative to the number of nodes.

General Pointer-Selection FLT. Let nodes be indexed $N_1, N_2, \ldots, N_i$ with $i \geq 1$. We fix a radix $b \geq 2$. For node N_i, we want to select predecessors from the interval $\{1, \ldots, i-1\}$ by filling each scale level with all valid multiples of b^n that still fit into the remaining suffix. We again work with "slices" of the index interval.

Define three sequences:

$$\begin{aligned} &(s_r)_{r\geq 1}, \\ &(\ell_r)_{r\geq 1}, \\ &(j_{r,p})_{r,p}. \end{aligned} \tag{20}$$

Initialization is identical to binary case (12) (13).

And for each step $r \geq 1$ such that $\ell_r > 0$ recursively execute the following actions:

1. Choose the largest scale that fits in the current slice:

$$n_r = \log_b(\ell_r), \quad u_r = b^{n_r}. \tag{21}$$

2. On this scale, consider the multiples $u_r, 2u_r, 3u_r, \ldots, (b-1)u_r$ and include all of them that still lie inside the slice length ℓ_r. Formally, for each $p \in \{1, \ldots, b-1\}$ with $pu_r \leq \ell_r$, define the global predecessor index:

$$j_{r,p_r^*} = s_r + pu_r - 1. \tag{22}$$

These are the pointers contributed by step r.

3. We then advance the slice to begin strictly after the last chosen pointer:

$$s_{r+1} = j_r + 1, \quad \ell_{r+1} = i - s_{r+1}. \tag{23}$$

We stop at the first $R = r$ such that $\ell_{R+1} = 0$. At that point, the entire interval $\{1, \ldots, i-1\}$ has been exhausted in a sequence of slices. Next comes the pointer-selection function. Let

$$m_i = \sum_{r=1}^{R} \left\{ \mathrm{p} \in \{1, \ldots, \mathrm{b}-1\} \mid \mathrm{pu}_r \leq \ell_r \right\}^{\#} \tag{24}$$

be the total number of pointers selected for node N_i. Enumerate all indices $j_{r,p}$ in lexicographic order on (r,p):

$$(\mathrm{j}_1, \mathrm{j}_2, \ldots, \mathrm{j}_{m_i}) = (\mathrm{j}_{1,1}, \mathrm{j}_{1,2}, \ldots, \mathrm{j}_{1,p_1^*}, \tag{25}$$

$$\mathrm{j}_{2,1}, \ldots, \mathrm{j}_{2,p_2^*}, \ldots, \mathrm{j}_{R,1}, \ldots, \mathrm{j}_{R,p_R^*}).$$

The general pointer-selection function is then defined as:

$$f_g(i,k) = \begin{cases} j_k, & 1 \leq k \leq m_i, \\ \textit{undefined}, & k > m_i. \end{cases} \tag{26}$$

The Loom aggregates these into the predecessor set:

$$P_g(i) = \bigcup_{k=1}^{m_i} \{ f_g(i,k) \}. \tag{27}$$

Both binary and general pointer-selection functions should add padding and place zero hashes for all missing pointers, since number of slots per level is pre-defined by radix of the formula, and each pointer has a respective one. This simplifies the algorithm of inclusion proof construction.

Summarizing, the obtained definition allows us to select any radix that we prefer, define how many pointer tree slots are needed for a certain number of supported nodes, and choose the desirable depth of the state tree needed to support some chain length. All the formulas for evaluation of complexity will be introduced in the next section.

PARAMETRIZATION AND STRUCTURAL PROPERTIES OF THE FLT

At this stage, the construction itself is considered fixed; the purpose here is not to introduce new mechanisms, but to formalize the design space exposed by FLT and to make explicit the tradeoffs that govern its configuration.

Radix and Pointer Fanout. Although any positive integer fits, we fix the k-ary traversal tree radix as

$$r = 2^a + 1, \quad a \in \mathrm{N}, \tag{28}$$

which determines the branching factor of the traversal structure. The reason behind such choice for the radix will be explained in the section about carry encoding. For each pointer node, the number of pointer slots consumed per scale level is

$$s = r - 1. \tag{29}$$

This reflects the fact that, at a given scale, all non-zero multiples of the base unit are populated. Increasing the radix increases the temporal span covered per level but simultaneously increases the per-node pointer density, imposing a practical upper bound on admissible values of r.

Supported Levels per Pointer Tree Depth. Let d denote the depth of the underlying pointer Merkle tree. Given that each level consumes (29) pointer slots, the total number of scale levels supported by a binary pointer tree of depth d is

$$l = \left\lfloor \frac{2^d}{s} \right\rfloor = \left\lfloor 2^{d-a} \right\rfloor. \quad (30)$$

This relation captures a central property of FLT: increasing the pointer tree depth increases the number of supported scale levels exponentially, while increasing the radix increases the temporal span represented by each level at the cost of higher pointer occupancy.

Node Capacity and Block Support. For a fixed radix exponent a and pointer-tree depth d, the total number of nodes supported by the pointer structure is

$$n(a,d) = r^l = \left(2^a + 1\right)^{\left\lfloor 2^{d-a} \right\rfloor}. \quad (31)$$

Let d_1 denote the depth of the state tree used for block inclusion proofs. In the baseline configuration, we take

$$d_1 = d, \quad (32)$$

so that traversal and state inclusion share the same depth budget. This may not be the case when state and pointer tree depths differ, yet simplifies the process of evaluation. Then, the total number of blocks supported by the combined pointer and state trees is

$$b = n(a,d) \cdot 2^{d_1} = \left(2^a + 1\right)^{\left\lfloor 2^{d-a} \right\rfloor} \cdot 2^{d_1}. \quad (33)$$

This expression makes explicit that FLT decouples supported history size from verification cost: the block span grows exponentially in the number of supported levels, while proof complexity depends only on the level traversed.

Hash Cost Model. Verification cost is measured in the number of hash computations required to validate a block-to-anchor proof. A single pointer hop requires

$$h_1 = d + 1 \quad (34)$$

hashes, corresponding to a Merkle proof through the pointer tree plus the root commitment. A block-in-node inclusion proof requires

$$h_2 = d_1 + 1, \quad (35)$$

which, under the baseline assumption (32), yields $h1 = h2$. This symmetry simplifies verifier logic and circuit construction. For a block located at pointer level $l_1 \in \{0, 1, \ldots, l\}$, the worst-case verification cost without additional caching is

$$w(l_1) = h_1 \cdot l_1 + h_2 = h_1 \cdot (l_1 + 1). \quad (36)$$

Temporal Coverage per Level. Each pointer level l_1 covers a contiguous range of blocks given by

$$B(l_1) \in \begin{cases} [0,\ d_1), & l_1 = 0 \\ \left[d_1 \cdot r^{l_1 - 1},\ d_1 \cdot r^{l_1}\right), & l_1 \geq 1 \end{cases}. \quad (37)$$

Thus, while the verification cost grows linearly with the level index l_1, the number of blocks covered by that level grows exponentially in r. This establishes the fundamental compression property of FLT: large temporal distances can be authenticated with a small, predictable increase in verifier effort.

Carry Encoding and Append Complexity. A key structural property of the FLT pointer tree is that the occupation of its leaf slots implicitly encodes the temporal position of nodes. This encoding emerges from the way pointers to nodes of different orders are organized within the pointer tree.

Each level of the pointer tree reserves a contiguous slot range for pointers referencing nodes of a particular order. When new nodes are appended, these slots are filled sequentially. Once all slots of a given level become occupied, the structure behaves analogously to a positional counter: the slots at that level are cleared, and their accumulated state is propagated to the next level through the creation of a higher-order node.

This process mirrors the carry operation in binary or radix-r counters. A sequence of appends typically updates only the lowest levels of the pointer tree, resulting in a stable and predictable append cost. However, when the slot range of a lower order becomes saturated, the carry propagation causes the corresponding leaves to be cleared and recommitted at a higher level of the tree. In naïve constructions, this would temporarily increase append cost due to recomputation of multiple leaves.

FLT mitigates this effect by exploiting the structured zoning of pointer slots. Pointers to nodes of different orders occupy slot ranges of size 2^n, which aligns with the natural carry behavior of positional counters. Because the leaves within such a range are guaranteed to be simultaneously cleared when a carry occurs, their commitment can be

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

represented by a single zero hash at the corresponding higher level of the pointer tree.

This observation allows the pointer tree to be implemented as a Carry Sparse Merkle Tree (Carry-SMT). Instead of recomputing commitments for each cleared leaf, the subtree corresponding to the cleared slot range is replaced by a single zero commitment. As a result, carry propagation does not increase append cost.

As a result, the append cost of an FLT node is composed of three bounded components:

– pointer tree construction cost (bounded by the pointer tree depth);

– state tree construction cost (bounded by the batch size);

– root merge cost (constant).

Under the obtained construction, the cost of appending a node and the cost of constructing the pointer tree for that node, which is bounded by the depth of the pointer tree, become equal. Since this depth is fixed for a given configuration, the append cost is constant.

When blocks are appended in batches, meaning a state tree commits multiple blocks into a single FLT node, this constant construction cost is amortized across all blocks in the batch. Consequently, the effective append cost per block becomes extremely small compared to conventional accumulator structures.

In summary, the pointer-slot occupation pattern of FLT forms an implicit positional counter, and by aligning pointer zoning with powers of two and representing cleared ranges with zero commitments, the structure achieves strictly constant *O(1)* append cost per node while preserving logarithmic traversal properties.

Island Nodes. The ordering rules of FLT create a class of nodes that occupy structurally advantageous positions within the traversal hierarchy. These nodes can be described using the expression

$$n = pr^k, \tag{38}$$

where r is the radix of the traversal tree, $1 \le p \le r-1$, $0 \le k \le l$, with l denoting the number of supported traversal levels.

Nodes satisfying this relation appear at positions that align with higher-order pointer ranges. Because traversal hops scale with powers of the radix, these nodes can be reached using fewer pointer hops than nodes located within the regular sequential append window.

Such nodes effectively form structural “islands” in the history of the FLT: checkpoints that provide shorter traversal routes across large segments of the structure.

The larger the exponent k, the greater the distance covered by the corresponding pointer hops and the fewer traversal steps required to reach the node. As a result, island nodes provide naturally efficient anchoring points for historical verification.

This property may have practical implications for systems that synchronize state with temporal lag, such as cross-domain verification or bridging mechanisms, where targeting island nodes may reduce proof size and verification effort.

Structural Pruning of Obsolete Traversal Commitments. The logarithmic traversal properties of FLT imply that multiple inclusion proof paths may exist for the same historical node. As the structure evolves, newly appended nodes introduce higher-order pointers that provide shorter traversal routes to older nodes.

Once a more efficient traversal path becomes available, previously valid proof paths may become permanently suboptimal and will no longer be selected by the traversal rule. Although the historical nodes themselves must remain available for verification, the intermediate commitment data corresponding to these obsolete paths no longer needs to be stored.

This observation enables structural pruning of redundant proof paths. Implementations may safely discard commitment data associated with traversal routes that will never again be chosen under the pointer selection function. Over time, this prevents the accumulation of unused path fragments that would otherwise inflate storage requirements.

Pruning therefore targets redundant traversal commitments rather than historical nodes, allowing the logical structure of the FLT to remain intact while significantly reducing the storage footprint of long-lived deployments.

Interpretation of the Design Space. Taken together, the parameters a, d, and d_1 define a tunable instrument rather than a single fixed data structure. Increasing the radix r shifts verification cost toward wider but fewer hops; increasing tree depth expands supported history while raising minimal inclusion cost. The explicit separation between pointer traversal and state inclusion allows these tradeoffs to be reasoned about independently.

Analysis of pointer tree changes over time allowed to apply both windowing optimization, and find a configuration where append cost becomes constant, which is a strong optimization for the anchor synchronization on the first phase of zk-bridges.

At this stage, no assumptions are made about anchor caching or radix-specific optimizations. Those mechanisms refine the cost model derived above but do not alter the structural relationships formalized here. Their impact is analyzed in the subsequent sections.

STRUCTURAL OPTIMIZATIONS OF FLT

Further structural optimizations are meant to refine the baseline cost model established in Section 5. They do not alter the correctness or security properties of the Fractal Loom Tree (FLT); instead, they reshape verification costs by exploiting temporal locality, structural asymmetry, and hardware-aware hashing tradeoffs. In particular, they significantly reduce the effective hash count for the block ranges that dominate real-world bridge usage.

Anchor Caching and Pointer Lifecycles. The primary optimization is anchor caching, which allows selected pointer nodes to be stored persistently in the verifying smart contract. Cached anchors are updated only when their corresponding pointer slots are overwritten, rather than on every block append. As a result, each cached pointer acquires a well-defined lifecycle.

For a pointer at level l_1, its lifecycle spans

$$L(l_1) = 2^{d_1} \cdot r^{(l_1+1)} \tag{39}$$

blocks. During this interval, the pointer remains valid and can be reused across all proofs referencing that level. This persistence eliminates the need to verify the initial pointer hop on every proof. Consequently, the worst-case verification cost derived in Section 5 can be rewritten as

$$w(l_1) = h_1 \cdot l_1, \tag{40}$$

which removes one full traversal hop from the cost model. This change preserves the logarithmic growth of verification cost with temporal distance while strictly lowering the constant factor for all non-zero levels.

Lazy Reorganizations. A distinctive property of FLT is that all nodes are appended sequentially and their commitments propagate through subsequent nodes. At first glance, this suggests that reorganizations could require large-scale recomputation across the structure, but FLT avoids this by introducing a simple restriction on traversal during inclusion proof verification.

The verification process begins with the selection of a cached anchor node from the most recent frontier of the structure. After this initial hop, subsequent traversal steps are restricted to nodes of strictly lower order.

Because pointer slots corresponding to each order occupy contiguous 2^n ranges within the pointer tree, the initial traversal direction can be identified deterministically, while all further traversal steps follow the decreasing-order rule.

This constraint introduces the notion of a lazy reorganization. When a reorganization occurs, the commitments corresponding to replaced nodes may still physically exist within later nodes of the structure, but they become unreachable under the traversal rule and therefore irrelevant for future proofs.

Only the cached anchors that intersect the reorganization path must be updated. Due to the lifecycle of cached anchors and the logarithmic pointer structure, the number of affected anchors remains small. In practical configurations, this corresponds to only a handful of nodes even in worst-case scenarios.

As a result, the computational cost of handling a reorganization becomes equivalent to the cost of verifying an inclusion proof, preserving the logarithmic operational bounds of the structure.

Layer-0 State-Only Caching. A further asymmetry can be exploited for the lowest levels of the structure. If the latest r nodes are not used as predecessors for subsequent nodes, their pointer trees are semantically inactive. In this case, only their state trees need to be cached.

This layer-0 cache allows the earliest blocks to be verified using state inclusion proofs alone, without any pointer traversal. For this prefix, verification cost collapses to a constant number of state-tree hashes, independent of temporal distance within the layer. This regime forms the lowest-cost verification path and establishes a constant-time fast path for the most recent blocks.

k-ary In-Node Trees. Both the pointer tree and the state tree may be generalized from binary Merkle trees to k-ary trees. Let

$$r_1 = log_2 k, \tag{41}$$

denote the radix exponent of these in-node trees. Increasing k reduces tree depth proportionally while increasing the number of words absorbed per hash. Under this generalization, the hash cost per traversal hop becomes

$$h_1 = 1 + \frac{d}{r_1}, \tag{42}$$

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

with an analogous expression for state inclusion. At the same time, each hash absorbs 2^{r1} words, increasing witness size but reducing the total number of hash invocations.

This tradeoff is particularly favorable for in-circuit verification in zk-STARK-based solutions [21]. While on-chain cost models weigh hash invocations and calldata words differently depending on the execution environment, zk circuits are typically dominated by hash constraint counts rather than word arity, though it comes with its own tradeoff. As a result, exchanging a small increase in word count for a reduction in hash invocations can lower prover workload and circuit size.

Combined Effect on Verification Cost. Taken together, these optimizations transform FLT from a purely logarithmic authenticated structure into a temporally stratified system with multiple verification regimes:

– a constant-cost regime enabled by state-only caching at layer 0;

– a shallow logarithmic regime dominated by cached anchors;

– a cold-history regime that retains logarithmic guarantees but benefits from high radix and reduced depth.

The introduction of anchor caching and k-ary trees preserves the exponential growth of block coverage per level. For each level l_1, the covered block range remains the same (37), but the effective verification cost is reduced according to the revised hash model. Thus, temporal span per unit of verification effort increases monotonically under these optimizations.

These effects are orthogonal to the baseline parameterization and can be applied selectively based on deployment constraints. Their combined impact on practical proof sizes and verification costs is demonstrated concretely in the following section.

CONCRETE CONFIGURATION AND COMPARATIVE EVALUATION

The objective of this section is not to claim asymptotic superiority, but to evaluate practical verification cost under realistic Ethereum bridge workloads, and compare them against the obtained results with existing accumulator data structures.

Ethereum Temporal Parameters. At the time of writing, Ethereum has exceeded 2^{24} blocks. Block production proceeds at approximately 2^{13} blocks per day, which implies roughly one month of block production per 2^{18} blocks [22]. To ensure long-term safety and to avoid premature exhaustion of the accumulator, we target support for at least 2^{26} blocks, well beyond the current chain height and typical bridge lookback windows.

Choice of Radix and Tree Depths

The state-tree depth is fixed at $d_1 = 6$, so each FLT node commits $2^6 = 64$ blocks. Therefore, supporting 2^{26} blocks requires at least 2^{20} FLT nodes. For each considered traversal radix, the binary pointer-tree depth is selected as the minimum depth that provides sufficient pointer capacity for this number of nodes. Each traversal scale consumes $r - 1$ pointer slots (23), and the resulting number of complete scale groups determines the supported node capacity.

The resulting configurations are presented in Table 1. The temporal scale cap denotes the maximum backward distance covered before traversal proceeds to the next scale.

The three configurations represent different combinations of append behavior, pointer-tree depth, and temporal span per level. For $r = 9$, each scale occupies eight pointer slots, so a $d = 6$ binary pointer tree provides eight complete scale groups. For $r = 17$, sixteen slots are consumed per scale, requiring a $d = 7$ pointer tree to support at least 2^{20} FLT nodes. In both cases, $r - 1$ is a power of two, so the scale groups remain aligned with complete binary subtrees and append behavior remains stable.

For $r = 11$, each scale consumes ten pointer slots. A $d = 6$ pointer tree therefore provides six complete scale groups within its 64 available slots, leaving four slots unused. This configuration increases the temporal span of each scale without increasing the pointer-tree depth, but its scale boundaries do not correspond to complete binary subtrees. Consequently, pointer cleanup is irregular and a stable append cost cannot be assumed.

The resulting append costs are 70 and 71 hash invocations per 64-block node for $r = 9$ and $r = 17$, respectively. The append cost of $r = 11$ is not evaluated because its pointer-slot groups are not aligned with the binary tree structure.

The larger radix also increases the temporal span covered by each traversal level. For $r = 17$, the first three scale caps correspond approximately to 3 hours, 2 days, and 1 month, respectively, under the assumed Ethereum block rate.

Table 1. **Evaluated FLT traversal configurations**

	Pointer tree parameters				*Temporal scale reach (blocks, time)*						*Append*
r	Slots / scale	Depth d	Scale groups	Node capacity	Scale 1		Scale 2		Scale 3		Cost / 64 blocks
9	8	6	8	43.05 M	512	1.5 h	4,096	12 h	32,768	4 d	70
11	10	6	6	1.77 M	640	1.9 h	6,400	19 h	64,000	7.8 d	–
17	16	7	8	6.98 B	1,024	3 h	16,384	2 d	262,144	32 d	71

Source: **compiled by the authors**

Verification Cost for *r = 17* Under Different In-Node Radices. This configuration is used for the verification-cost comparison because its first two temporal regimes cover the selected recency ranges. The state and pointer trees are evaluated with binary, quaternary, and octary in-node hashing.

For the upper verification layer, the node commitment is reconstructed from two pointer-tree peaks and the state-tree root. The hashes required to obtain these components and the final hash combining them are included in the reported totals. Thus, the use of a wider in-node radix reduces the number of tree levels without adding another terminal hash.

The resulting costs are *6*, *15*, and *23* hash invocations for binary in-node trees, *3*, *8*, and *12* for quaternary trees, and *2*, *6*, and *9* for octary trees across the three temporal regimes. The first value in each group comes from layer-0 caching optimization.

Comparison with MMR and MMB. For comparison in Table 2, the recency values *k* are taken from the external MMR/MMB cost analysis [23]. The source assumes an item production interval of *6* seconds, so the comparison is made using the same recency values *k* rather than by converting them to the Ethereum time intervals used above.

MMR is used as the primary baseline, while MMB is included as an additional reference. The FLT values are determined directly from the corresponding temporal regimes of the *r = 17* configuration.

Comparative Summary. The choice of radix demonstrates a clean tradeoff. *r = 9* provides stable append behavior and lower pointer-tree depth, at the cost of a smaller temporal span per level. The choice of *r = 11* provides lower pointer-tree depth and a larger temporal span per level, at the cost of unstable append behavior. The choice of *r = 17* provides stable append behavior and a larger temporal span per level, at the cost of a higher pointer-tree depth.

For the direct comparison with existing binary Merkle-based accumulators, the *r = 17* binary FLT configuration is the primary case. Against MMR the optimization ranges between *19.6%* and *66.0%*, while against MMB − between *2.7%* and *52.6%*. The main reduction for binary FLT is obtained for the most recent elements, while the cost remains lower over the larger recency ranges considered.

Table 2. **FLT verification cost compared with MMR and MMB**

			FLT hashes / reduction (%) per r_1					
	Recency k (blocks)	*Baseline hashes*	Binary (r_1=1)		Quaternary (r_1=2)		Octary (r_1=3)	
MMR	600	17.65	6	66.0 %	3	83.0 %	2	88.7 %
	2,400	18.66	15	19.6 %	8	57.1 %	6	67.8 %
	14,400	19.95	15	24.8 %	8	59.9 %	6	69.9 %
MMB	600	12.67	6	52.6 %	3	76.3 %	2	84.2 %
	2,400	15.41	15	2.7 %	8	48.1 %	6	61.1 %
	14,400	18.97	15	20.9 %	8	57.8 %	6	68.4 %

Source: **compiled by the authors based on [14], [23]**

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

The quaternary and octary cases further reduce the number of hash invocations, with reductions relative to MMR spanning *59.9%* to *83.0%* and *69.9%* to *88.7%* respectively. These cases therefore show the potential benefit of increasing in-node arity for in-circuit verification, while the binary case provides the most direct comparison with the existing accumulator structures.

The obtained configurations consequently provide different combinations of append stability, temporal coverage, pointer-tree depth, and verification cost, allowing these properties to be adjusted according to the requirements of the target workload.

CONCLUSIONS

As a result of the conducted research, a number of conclusions can be drawn, corresponding to the objectives and analytical stages of the paper:

1. It has been established that proof cost for accumulator-based verification in zk-bridge architectures is most appropriately measured in terms of cryptographic hash operations and their input word counts, rather than asymptotic tree depth alone. This cost model more accurately reflects both on-chain verification expenses and in-circuit prover workload under realistic deployment conditions.

2. The analysis of existing accumulator data structures, including Merkle Mountain Ranges and their variants, demonstrates that while they provide strong asymptotic guarantees, they are not well suited to verification workloads with pronounced temporal locality. In particular, their proof size distributions remain largely insensitive to the recency of verified elements, and their structural properties inhibit efficient caching and reuse of intermediate states.

3. Second-order cryptographic accumulator concept, and the Fractal Merkle Tree (FMT) and Fractal Loom Tree (FLT) data structures have been formally defined and analyzed. These constructions generalize classical Merkle accumulators by introducing multi-scale pointer selection, enabling verification costs that grow logarithmically with the distance of a block from the most recent state. This establishes FLT as a temporally adaptive accumulator.

4. The parameter space of FLT has been characterized, including radix, pointer slot allocation, tree depths, and derived capacity bounds. It has been shown that these parameters allow fine-grained control over both the minimal verification cost and the span of cheaply verifiable history, making it possible to tailor the data structure to specific workload assumptions.

5. Anchor caching has beenS introduced as an optimization technique that exploits the internal structure of FLT. By persisting selected pointer and state tree nodes across verification epochs, anchor caching reduces worst-case proof sizes by eliminating redundant traversal steps and enables predictable pointer lifecycles. Analysis of structural properties additionally unveiled the properties of constant append cost, special nodes with cheaper proofs, and optimization of obsolete paths. These optimizations further amplify the benefits of temporal locality and are particularly effective in zk-circuit settings, where trading additional storage for reduced recomputation, and achievement of fixed logic without loops yields substantial efficiency gains.

6. A concrete parameterization targeting large-scale blockchain histories has been presented and evaluated. Using realistic Ethereum block production rates, the selected configuration demonstrates a significant reduction in proof size across relevant verification horizons when compared to existing accumulator designs. The results confirm that the proposed approach strictly dominates Merkle Mountain Range–based constructions under requirements of zk-bridges.

Thus, all declared objectives of the study have been achieved, and the stated purpose of the research has been fulfilled: the concept of second-order cryptographic accumulators has been established, and a derived cryptographic accumulator data structure has been developed and analyzed, providing the lowest expected proof size under realistic, temporally biased verification workloads.

Further research directions include the exploration of alternative commitment primitives in place of cryptographic hashing, such as algebraic and polynomial commitment schemes; the investigation of dynamic and adaptive parameter selection based on observed access patterns; the extension of the presented constructions to more general directed acyclic graph topologies; and the integration of pruning, sliding-window techniques, and positional encoding to further optimize storage and verification efficiency.

Conflict of interests: Authors declare that they do not have conflict of interests in financial, personal, copyright or any other nature, which could have influenced the research, as well as the results published in this article

Financing: Research was carried out without financial support

Accessibility data: This study is theoretical and did not involve the collection or analysis of external datasets. All data and materials necessary to understand and verify the reported results are available in digital or graphical form within the manuscript

Using artificial intelligence tools: During the preparation of this manuscript, the authors used ChatGPT for language editing and text refinement. The authors reviewed and edited the generated output and take full responsibility for the content of the publication

REFERENCES

1. Yin, R., Yan, Z., Liang, X., Xie, H. & Wan, Z. “A survey on privacy preservation techniques for blockchain interoperability”. *Journal of Systems Architecture*, 2023; Vol. 140: 102892. DOI: https://doi.org/10.1016/j.sysarc.2023.102892.

2. Belchior, R., Süßenguth, J., Feng, Q., Hardjono, T., Vasconcelos, A. & Correia, M. “A Brief History of Blockchain Interoperability”. *Communications of the ACM*. 2024; Vol. 67, No. 10: 62–69. DOI: https://doi.org/10.1145/3648607. https://www.scopus.com/pages/publications/85204046027.

3. Xie, T., Zhang, J., Cheng, Z., Zhang, F., Zhang, Y., Jia, Y., Boneh, D. & Song, D. “zkBridge: Trustless Cross-chain Bridges Made Practical”. *arXiv preprint*, 2022. DOI: https://doi.org/10.48550/arXiv.2210.00264.

4. Patel, O. “Blockchain Layer 2 Rollups, Optimistic vs Zero Knowledge”. *International Journal of Advanced Research in Engineering and Technology*. 2023; Vol. 14, No. 2: 13–29. DOI: https://doi.org/10.34218/IJARET.14.2.002.

5. Ahmadvand, M. & Souto, P. “Towards Cost-Effective ZK-Rollups: Modeling and Optimization of Proving Infrastructure”. *arXiv preprint*, 2025. DOI: https://doi.org/10.48550/arXiv.2509.16581.

6. Liu, H., X. Luo, X., Liu, X. & Xia, X. “Merkle Tree: A Fundamental Component of Blockchains”. *International Conference on Electronic Information Engineering and Computer Science (EIECS)*. Changchun, China, 2021. p. 556-561. DOI: https://doi.org/10.1109/EIECS53707.2021.9588047.

7. Cassez, F. “Verification of the Incremental Merkle Tree Algorithm with Dafny”. *arXiv preprint*, 2021. DOI: https://doi.org/10.48550/arXiv.2105.06009.

8. Kuznetsov, O., Frontoni, E., Kuznetsova, K. & Arnesano, M. “Optimizing merkle proof size through path length analysis: a probabilistic framework for efficient blockchain state verification”. *Future Internet*, 2025; Vol. 17, No. 2: 72, https://www.scopus.com/pages/publications/85218462147. DOI: https://doi.org/10.3390/fi17020072.

9. Kate, A., Zaverucha, G. M. & Goldberg, I. “Constant-Size commitments to polynomials and their applications”. *Advances in Cryptology – ASIACRYPT 2010, Lecture Notes in Computer Science*. 2010; Vol. 6477: 177–194. DOI: https://doi.org/10.1007/978-3-642-17373-8_11.

10. Zhou, Z., Zhang, Z., Liu, J. & Qian, H. “Efficient Inner Product Arguments with Sublogarithmic Proof and Sub-Square-Root Verifier. *Cybersecurity”.* 2025; Vol. 8: 45, https://www.scopus.com/pages/publications/105010174616. DOI: https://doi.org/10.1186/s42400-024-00304-x.

11. Kemmoe, V. Y. & Lysyanskaya, A. “RSA-Based Dynamic Accumulator without Hashing into Primes”. *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*, 2024. p. 4271–4285, https://www.scopus.com/pages/publications/85215531057. DOI: https://doi.org/10.1145/3658644.3690199.

12. Bonneau, J., Chen, J., Christ, M. & Karantaidou, I. “Merkle Mountain Ranges are Optimal: On Witness Update Frequency for Cryptographic Accumulators”. *Advances in Cryptology – CRYPTO 2025*. 2025. pp. 170–202, https://www.scopus.com/pages/publications/105014161291 DOI: https://doi.org/10.1007/978-3-032-01878-6_6.

13. Liang, C., Zhang, J., Ma, S., Zhou, Y., Hong, Z., Fang, J., Zhou, Y. & Tang, H. “Study on Data Storage and Verification Methods Based on Improved Merkle Mountain Range in IoT Scenarios”. *Journal of King Saud University – Computer and Information Sciences*. 2024; Vol. 36, No. 6: 102117, https://www.scopus.com/pages/publications/85198378294. DOI: https://doi.org/10.1016/j.jksuci.2024.102117.

14. Cevallos, A., Hambrock, R. & Stewart, A. “The Merkle Mountain Belt”. *arXiv preprint*, 2025. DOI: https://doi.org/10.48550/arXiv.2511.13582.

15. Campanelli, M., Fiore, D., Han, S., Kim, J., Kolonelos, D. & Oh, H. “Succinct Zero-Knowledge Batch Proofs for Set Accumulators”. *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2022. p. 455–469, https://www.scopus.com/pages/publications/85143054775 DOI: https://doi.org/10.1145/3548606.3560677.

16. Wang, W., Li, X. & Zhao, H. “DCAF: Dynamic Cross-Chain Anchoring Framework using Smart Contracts”. *The Computer Journal”.* 2022; Vol. 65, No. 8: 2164–2182, https://www.scopus.com/pages/publications/85156166793. DOI: https://doi.org/10.1093/comjnl/bxab052.

17. Agrawal, S., Neu, J., Tas, E. N. & Zindros, D. “Proofs of Proof-of-Stake with Sublinear Complexity”. *arXiv preprint*, 2022. DOI: https://doi.org/10.48550/arXiv.2209.08673.

18. Kirejczyk, M., Kalka, M. & Logvinov, L. “Historical and Multichain Storage Proofs”. *arXiv preprint*, 2024. DOI: https://doi.org/10.48550/arXiv.2411.00193.

19. Belchior, R., Vasconcelos, A., Guerreiro, S. & Correia, M. “A Survey on Blockchain Interoperability”: *Past, Present, and Future Trends*. *ACM Computing Surveys*. 2021; Vol. 54, No. 8: 168. 1–41. DOI: https://doi.org/10.1145/3471140.

20. Li, F. & McMillin, B. “A Survey on Zero-Knowledge Proofs”. *Advances in Computers*. 2014; Vol. 94: 25–69. DOI: https://doi.org/10.1016/B978-0-12-800161-5.00002-5.

21. Arade, M. S. & Pise, N. N. “ZK-STARK: Mathematical Foundations and Applications in Blockchain Supply Chain Privacy”. *Cybernetics and Information Technologies*. 2025; Vol. 25, No. 1: 3–18. DOI: https://doi.org/10.2478/cait-2025-0001.

22. Kapengut, E. & Mizrach, B. “An Event Study of the Ethereum Transition to Proof-of-Stake”. *Commodities*. 2023; Vol. 2, No. 2: 96–110. DOI: https://doi.org/10.3390/commodities2020006.

23. Cevallos, A. & Hambrock, R. “Merkle Mountain Belt: Cost Saving Analysis + Funding Proposal V3”. *HackMD*, 2025. – Available: https://hackmd.io/@MerkleMountainBelts/MMB-Funding-Proposal-V3. – [Accessed: 21.09.2026].

Received 25.08.2026
Received after revision 20.10.2026
Accepted 28.10.2026

DOI: https://doi.org/10.15276/hait.9.2026.32
UDC 004.021:004.422.63:004.056.55

Фрактальні принципи побудови криптографічних акумуляторів другого порядку

Бучіхін Артем Андрійович[1)]
ORCID: https://orcid.org/0009-0006-9562-1273; artembuchihin@gmail.com
Болтьонков Віктор Олексійович[1)]
ORCID: https://orcid.org/0000-0003-3366-974X; vaboltenkov@gmail.com. Scopus Author ID: 57203623617
[1)] Національний університет «Одеська Політехніка», пр. Шевченка, 1. Одеса, 65044, Україна

АНОТАЦІЯ

Криптографічні акумулятори є ключовим компонентом систем верифікації блокчейнів, включаючи мостові рішення з нульовим розголошенням (zero-knowledge bridges), де розмір доказу безпосередньо впливає на витрати в мережі та продуктивність докажчика. Існуючі конструкції акумуляторів, такі як Merkle Mountain Range та його варіанти, забезпечують

сильні асимптотичні гарантії, але не повністю враховують часово-зумовлені шаблони доступу, характерні для справжніх навантажень верифікатора при роботі. **Метою роботи** є представлення та аналіз нової криптографічної акумуляторної структури даних, яка мінімізує очікуваний розмір доказу за умов справжніх, часово-зміщених навантажень верифікації. У роботі запропоновано та проаналізовано новий клас акумуляторних структур даних – Fractal Merkle Tree (FMT) та Fractal Loom Tree (FLT), з метою мінімізації очікуваного розміру доказу під такими навантаженнями. Вартість доказу формалізується через криптографічні хеш-операції та кількість вхідних слів, що відображає як витрати на верифікацію в мережі, так і навантаження докажчика всередині zk-ланцюга. Конструкція Fractal Loom Tree використовує багатошарове вибирання вказівників (multi-scale pointer selection), що забезпечує зростання вартості верифікації логарифмічно залежно від часової відстані від найсвіжішого стану. Характеризується простір параметрів Fractal Loom Tree, показуючи, як радікс, глибина дерева та розподіл вказівників

впливають на часово-залежний розподіл розмірів доказів. Запропоновано оптимізацію кешування якорів, що дозволяє повторно використовувати стійкі проміжні стани та додатково знижує витрати верифікації у найгіршому випадку. Наведено напрямки подальших досліджень для розширення конструкції на альтернативні примітиви зобов'язання, адаптивний вибір параметрів та більш загальні графові структури. Основним результатом роботи є розробка криптографічного акумулятора, який забезпечує суттєво менші докази порівняно з існуючими конструкціями під реалістичними, часово-зумовленими навантаженнями блокчейн-верифікації.

Ключові слова: криптографічні акумулятори; дерева Меркла; докази з нульовим розголошенням; zk-містки; верифікація блокчейну; оптимізація розміру доказу; зобов'язуючі структури даних; верифіковні структури даних

ABOUT THE AUTHORS



Artem A. Buchikhin - PhD Student, Department of Information Systems, Odesa Polytechnic National University, 1, Shevchenko Ave. Odesa, 65044, Ukraine
ORCID: https://orcid.org/0009-0006-9562-1273; artembuchihin@gmail.com
***Research field*:** Blockchain technologies

Бучіхін Артем Андрійович - аспірант, кафедра Інформаційних систем, Національний університет «Одеська політехніка», пр. Шевченка, 1. Одеса, 65044, Україна
Галузь досліджень: блокчейн технологии



Viktor O. Boltenkov - PhD, Associate Professor, Information System Department. Odesa Polytechnic National University, 1, Shevchenko Ave. Odesa, 65044, Ukraine
ORCID: https://orcid.org/0000-0003-2777-3137; vaboltenkov@gmail.com. Scopus Author ID: 57203623617
***Research field*:** Blockchain technologies; signal processing

Болтьонков Віктор Олексійович - кандидат технічних наук, доцент кафедри Інформаційних систем. Національний університет «Одеська політехніка», пр. Шевченка, 1. Одеса, 65044, Україна
Галузь досліджень: блокчейн технологии, обробка сигналів