Smaran: Serving Authenticated Time-Travel Queries

Shistata Subedi[*], *Asim Nepal*[*], *Shubham Mishra*[†], *Suyash Gupta, Aniket Kate*[◇], *Hein Meling*[‡]
University of Oregon, [†]*University of California Berkeley,* [◇]*Purdue University,* [‡]*University of Stavanger*

Abstract

Time-travel queries ask what data looked like at a past time or over an interval. Modern databases support time-travel queries, but this support is insufficient for applications that cannot trust the query server, such as secure messaging and decentralized applications, where clients need proofs that the returned results are correct. Existing approaches either require aggregating evidence from multiple auditors, incur per-step proof costs that scale linearly with history length, or impose massive storage and communication costs.

In this paper, we introduce Smaran, an authenticated data structure that enables servers to answer time-travel queries efficiently with a constant number of fixed-size proofs, independent of the query range. Smaran achieves this through a novel integration of segment trees with vector commitments, sidestepping scalability challenges as history grows via a hierarchical architecture, and avoiding redundancy by introducing versioned vector commitments.

1 Introduction

In this paper, we present **Smaran**, a system that facilitates efficient authenticated *time-travel queries*.

Time-travel queries ask what the system's data looked like in the past: what a record was at a specific point in time, or how it evolved over an interval. For example, clients may review their bank accounts to access monthly or annual credit card statements, while auditors may perform time-travel queries to investigate incidents and perform audits at a company. To meet this demand, modern databases provide support for such queries [2, 15, 17, 44, 51], which is sufficient for applications that can *trust* the database server and/or the cloud provider to maintain the integrity of past data.

Unfortunately, no efficient solution exists for time-travel queries in systems where the query server cannot be trusted. In this model, no single entity is responsible for verifying transactions or maintaining and attesting to history; instead, trust is established collectively through cryptographic proofs and agreement among participants. For instance, secure messaging services (e.g., WhatsApp and iMessage) adopt *key transparency*: a server stores public keys in a public append-only log, and clients authenticate their integrity through independent auditors [35, 41]. This requires preserving key history, as many security questions are inherently retrospective, e.g., which key was associated with Bob when Alice sent a particular message. Without preserving this history, a malicious server could present different key versions to different users at different times. Similarly, secure software supply-chain services [49, 50] adopt *signature transparency* by recording software signing events in an append-only log [48], so auditors can verify that artifacts are publicly visible and consistent. More broadly, decentralized application platforms [18, 38] and permissioned ledger systems [47] provide immutable, auditable ledgers maintained collectively by mutually distrusting parties. Querying historical ledger state is a core requirement for such ledgers; users routinely need to retrieve past information, including annual transaction statements, and specific historical balance versions. Consequently, these applications rely on third-party services [3] to provide clients with access to transaction histories because efficiently supporting time-travel queries poses several challenges:

1. *High Computational Costs.* To answer a time-travel query, the server must return not only the requested values but also proof of their correctness. In existing systems, this typically requires generating evidence for each operation (for example, one proof per transaction), so the server's work grows linearly with the length of the requested interval [18, 42]. Some prior approaches incur additional overhead because they reconstruct past states by replaying long sequences of updates before producing a proof [39].
2. *Communication Bottleneck.* Additionally, existing systems [18, 42] must return all individual proofs to the client; for larger range queries, the response size grows proportionally, which can bottleneck network bandwidth.
3. *Reliance on Multiple Auditors.* Alternatively, systems can replace publicly verifiable proofs with client-side queries to $f+1$ independent auditors to tolerate up to f adversarial parties [33, 43], incurring substantial communication costs, as f is typically large in decentralized systems.

These challenges expose a core limitation of existing designs: supporting authenticated time-travel queries incurs prohibitive overheads. Smaran addresses these challenges by introducing a novel data structure purpose-built for authenticated time-travel queries. Smaran enables a server to answer such queries with a constant number of fixed-size proofs, independent of the query range or historical depth. This property eliminates client-side proof aggregation and removes both bandwidth and computation bottlenecks,

making verifiable time-travel queries finally practical.

Smaran achieves this efficiency through a novel integration of segment trees [16] with cryptographic vector commitments [30]. Segment trees provide a natural way to organize and aggregate data across arbitrary time ranges, while vector commitments supply succinct, verifiable proofs for the aggregated values. Combined, these two primitives yield an authenticated range index that can attest to large historical intervals with compact, constant-size proofs.

A naive integration of these primitives, however, quickly encounters two barriers: scalability and redundancy. First, vector commitments have bounded capacity, so a single commitment cannot efficiently cover an ever-growing timeline. A natural fix might be to stack commitments hierarchically, letting higher layers commit to the outputs of lower ones. But this straightforward hierarchy reverses the very efficiency we seek: a range that spans many lower-layer commitments would require one proof per segment at every level. Smaran breaks this pattern by summarizing entire lower-layer trees within single upper-layer leaves, allowing one proof on an internal node to simultaneously certify large spans of history.

Second, existing commitment schemes implicitly treat every time step (or *epoch*) as a new version, even when no updates occur [4]. This approach faithfully encodes change but wastes space and time for epochs where the state remains unchanged. Smaran eliminates this redundancy by introducing native versioning directly into the commitment layer. Each commitment natively reflects only versioned updates, preserving succinctness even across sparse or inactive periods.

To demonstrate Smaran's effectiveness, we integrate it into two representative settings: key transparency and decentralized ledgers. First, on a workload of 10k users with up to 2047 key versions, Smaran outperforms state-of-the-art frameworks like CONIKS [41] and OPTIKS [35], boosting time-travel query throughput by 5.7× and reducing latency by 82.5%, all while cutting communication overhead by 95.2%. Second, when applied to decentralized ledgers to answer time-travel queries spanning 2.6 million Ethereum blocks from 2024, Smaran achieves up to 589× higher throughput and 99.94% lower latency compared with alternative authenticated structures such as Merkle trees [42], Verkle trees [9], and Cauchy proofs [39].

In summary, we make the following contributions:

- We present Smaran, a novel authenticated data structure that enables authenticated time-travel queries.
- We show how to answer queries with a constant number of fixed-size proofs, independent of the query range.
- We implement Smaran in two representative applications: decentralized ledger and key transparency to enable high-performance time-travel queries.

2 The Need for Time-Travel Queries

Time-travel queries ask how data looked at a specific time in the past or how it evolved over an interval. They play a critical role in auditing financial records, investigating incidents, debugging versioned systems [55], and demonstrating regulatory compliance by reconstructing past states exactly as they appeared at transaction time. Modern databases support time-travel queries, but their mechanisms fail when clients cannot trust the query server. In such environments, clients must not only retrieve historical data but also verify its correctness using cryptographic proofs.

Key Transparency. Secure messaging services such as WhatsApp and iMessage–which implement *key transparency*–exemplify applications where clients cannot rely solely on a single server for data integrity. Key transparency addresses the risk that a server/provider might equivocate about a client's public keys. Consider Alice and Bob using WhatsApp: when Bob registers, the server stores his public key and later returns it to Alice when she wants to send him an end-to-end encrypted message. Without key transparency, Alice must fully trust the server to return the correct key; a compromised server could instead provide an attacker-controlled key, enabling an undetectable man-in-the-middle attack.

One way to mitigate this risk is by recording all identity–key bindings (for example, Bob's phone number to his public key) in a public append-only log monitored by independent auditors, so that clients can verify both that the returned key appears in the log and that the log has not been rewritten. Moreover, a client's public key is not treated as a single, permanent value but as an evolving sequence of keys tied to the same identity, changing when the client gets a new device, re-installs the app, or when the protocol rotates keys for security. Thus, persisting this key history is fundamental to key transparency, because many security questions (time-travel queries) are inherently about the past (e.g., which key was associated with Bob when Alice sent a particular message). Prior key transparency papers treat these time-travel queries as monitoring queries: to prove that Bob's key has α versions, the monitoring query must show the integrity of versions 1 through α and non-existence of version $\alpha+1$.

Decentralized Queries. Decentralized systems such as Ethereum [18] and Stellar [38] provide tamper-evident, transparent ledgers maintained collectively by mutually distrusting nodes, without relying on a central authority. A core property of such systems is immutability: once recorded, ledger data is preserved indefinitely, which makes historical queries a first-class requirement rather than an afterthought. Typical queries include checking an account's current balance, obtaining annual transaction statements, or retrieving the balance at specific past points in time. Consider Alice, an Ethereum user who requests her annual account statement. The statement should enumerate all transactions involving her account and, crucially, provide a cryptographic proof for each entry. Using these proofs, Alice should be able to verify that every transaction listed actually appears in the ledger and that aggregating their effects reproduces the publicly visible balance of her account. In an ideal design, Alice needs only to contact a sin-

gle node in the network: that node returns the statement plus proofs, and Alice independently verifies correctness, without querying a majority of nodes. This is only safe if any answer returned by an arbitrary node is self-verifiable–that is, Alice can detect incorrect or fabricated data even when the responding node is malicious.

3 System Model

We assume a system with two types of participants: *clients* and a *server*. Each client has an associated *record*, identified by k, whose updates produce a sequence of versions. The server maintains a directory mapping each identifier k to the version history of the corresponding record.

Public Ledger. The server generates *commitments* to the system state; a commitment is a cryptographic digest that binds a set of values [30]. These commitments are then recorded in a publicly verifiable ledger (or log) that defines a global order of updates. The ledger enables clients to independently authenticate record histories.

Commitment Generation and Failure Model. Smaran requires that commitments are computed and disseminated honestly, but treats the mechanism that ensures this as a black box. Example mechanisms include Byzantine agreement among independent parties [7, 10], Trusted Execution Environments [1, 26, 27], and Byzantine fault-tolerant public ledgers [18, 33, 43]. Smaran inherits the failure model of whichever mechanism is deployed.

Smaran assumes that a malicious server cannot break standard cryptographic primitives or forge valid commitments.

Formalizing Authenticated Time-Travel Queries. Each record k evolves through a sequence of versions over time, represented as $\langle v_1, v_2, \ldots, v_n \rangle$, where v_i denotes the value of the record at time t_i. A client can issue a time-travel (range) query that requests the versions of record k whose timestamps fall within a given interval $[t_i : t_j]$. In response, the server returns the corresponding sequence of versions $\langle v_i, \ldots, v_j \rangle$, together with a *witness* w attesting to their correctness and completeness. A witness is a short proof that a specific value belongs to the committed set.

An authenticated time-travel query system should satisfy the following properties:

Succinctness. The proof size is constant and independent of the queried range.

Efficiency. Proof generation and verification costs grow sub-linearly with history length.

Robustness. Proofs are publicly verifiable, so querying a single server is sufficient.

Generality. Proofs remain correct and compact for any record type and arbitrarily long histories.

4 Towards Authenticated Time-Travel Queries

Existing authenticated data structures are designed primarily for point queries, where a single proof suffices to authenticate one value. Time-travel queries, by contrast, require proofs over intervals, where no existing approach is both efficient and scalable. In this section, we survey three candidate primitives—Merkle trees, vector commitments, and segment trees—and identify the specific limitations that make them insufficient for authenticated time-travel queries.

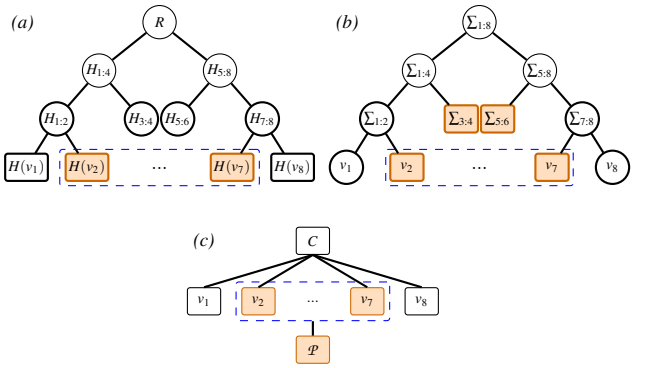


Figure 1: Data structures for authenticated time-travel queries. (a) Merkle tree. (b) Segment tree. (c) Vector commitments.

4.1 Merkle Trees

A Merkle tree [42] is a classic structure for proving that a specific value is included in a committed dataset. Given an ordered list of values $\langle v_1, v_2, \ldots, v_n \rangle$, a Merkle tree arranges them in a complete binary tree, where each leaf stores the hash of one value v_i, and each internal node stores the hash of its two children's hashes. The root hash R then serves as a commitment to the entire list. *Note.* To ensure the cryptographic security of the root hash, we require a collision-resistant hash function [31].

To prove inclusion of a value v_i, the prover returns a *witness*—a set of sibling hashes along the path from leaf i to the root. Together with v_i, this witness allows the verifier to recompute the hashes bottom-up and check that the result matches the committed root. Thus, for a tree of height h and hash size b bits, the witness size is $h \times b$.

Merkle trees support point queries efficiently, but extending them to range queries is costly. To answer a range query over $\langle v_i, \ldots, v_j \rangle$, the prover must produce a separate witness for each value in the range. Thus, for a range spanning n versions, the witness size and verification cost grow to $O(n \times h)$.

Example. Figure 1a illustrates a Merkle tree over 8 versions $\langle v_1, \ldots, v_8 \rangle$. Each internal node stores the hash of its two children's concatenation; for example, $H_{1:2} = H(H(v_1) \| H(v_2))$. To answer a range query over $\langle v_2, \ldots, v_7 \rangle$, the prover must produce a separate witness for each value in the range. For v_2, this witness consists of one sibling hash per level, $H(v_1)$, $H_{3:4}$, and $H_{5:8}$, totaling 3 hashes for a tree of height 3. Repeating this for all six values yields 18 hashes in total, confirming the $O(n \times h)$ cost.

4.2 Vector Commitment Schemes

Unlike Merkle trees, vector commitments [30, 37, 46] allow committing to a list of values once, and later proving the value at any position i with a single, fixed-size proof. Specifically,

we encode the versions of record k as a vector $\langle v_1, v_2, ..., v_n \rangle$ and represent this vector as a polynomial $P(x)$ satisfying $P(i) = v_i$ for all $i \in [1, n]$. This commitment is a single point on an elliptic curve [30] that acts as a compact summary of all values in the vector, computed deterministically from P's coefficients and a set of public precomputed curve points. This allows anyone to later verify inclusion of any individual entry without accessing the entire vector.

At first glance, vector commitments appear promising for range queries: committing once to all versions of record k yields a witness of constant size, independent of the query range. However, this approach has two critical limitations. First, proving a batch of positions requires dividing the committed polynomial by a vanishing polynomial whose degree equals the batch size. For a range spanning n versions, this results in $O(n)$ cost for both proof generation and verification. Unlike Merkle trees, where each operation is a hash computation, each step here involves an elliptic curve operation–typically orders of magnitude slower. Second, the maximum vector length, L, is fixed during system setup. For example, the curve BLS12-381 typically limits vectors to 4096 elements [21], restricting how many versions a single commitment can encode.

Example. Figure 1c illustrates a vector commitment C over 8 versions $\langle v_1, ..., v_8 \rangle$ of a record. To answer a range query over $\langle v_2, ..., v_7 \rangle$, the prover constructs a single constant-size witness $\mathcal{P}$. However, doing so requires 6 polynomial divisions, one per queried position.

4.3 Segment Trees

Both limitations of vector commitments—costly proof generation and bounded vector size—motivate us to consider segment trees, a classic data structure for efficient range aggregation. A *segment tree* precomputes summaries over sub-intervals of an ordered list, so that any range query can be answered by combining a small number of these summaries.

Concretely, given an ordered list $\langle v_1, v_2, \ldots, v_n \rangle$ and an associative function f, a segment tree organizes the list in a complete binary tree, where each internal node stores the value of f applied to all elements in its sub-range. This allows any range query $f(v_i, ..., v_j)$ to be answered in $O(\log n)$ time, by aggregating precomputed summaries from a small set of nodes that jointly cover $[i, j]$, touching only a constant number of nodes per level.

Example. Figure 1b illustrates a segment tree over 8 versions $\langle v_1, ..., v_8 \rangle$ using summation ($\sum$) as f, where $\sum_{1:2}$ denotes the sum of v_1 and v_2, and root $\sum_{1:8}$ is the sum of all leaves. A range query over $\langle v_2, ..., v_7 \rangle$ requires combining four precomputed nodes: v_2, $\sum_{3:4}$, $\sum_{5:6}$, and v_7.

Answering time-travel queries. Segment trees can efficiently answer range queries. However, associative functions such as sum, count, and max are lossy: many distinct input sequences produce the same aggregate. For example, the sum of two numbers can be represented in different ways: $5 = 2 + 3$, or $5 = 4 + 1$. As a result, committing to an aggregate does not commit to the individual versions it was computed from. This makes such functions insufficient for *authenticated* time-travel queries (§3), which require each returned version to be verifiably correct and complete. To support authentication, the function f must instead produce a deterministic cryptographic binding that uniquely commits to the exact set of versions in each node's sub-range.

While cryptographic accumulators [6, 29, 32, 53] can serve as the binding function, a simpler approach suffices: computing each internal node as the hash of its two children, exactly as in a Merkle tree. This produces a cryptographic binding at every level of the segment tree. The key challenge is that neither Merkle trees nor vector commitments can serve as a scalable commitment layer on their own. Merkle trees produce proofs that grow with the range, and vector commitments cannot scale to long histories. What is needed is a construction that produces bindings at every node and combines them into a single compact commitment. Smaran achieves this by combining such a Merkleized segment tree with vector commitments, as we describe next.

5 Smaran

Smaran delivers authenticated time-travel queries while satisfying four key properties: succinctness, efficiency, robustness, and generality (§3). Yet achieving this demands an architecture that seamlessly supports the query lifecycle: data ingestion, query processing, and response verification.

The data ingestion phase is most critical: it defines the data layout, which directly impacts time-travel query performance. Modern databases offer elegant solutions but cannot operate with an untrusted query server–where responses must include verifiable proofs. Existing primitives (§4) that support generating such proofs fail to meet three of four key properties (succinctness, efficiency, generality).

Smaran achieves efficient query processing through structured data maintenance that accelerates compact proof generation. Specifically, Smaran combines a segment tree–whose internal nodes are Merkle hashes–with vector commitments into a structure we call the **Merkleized Segment Tree (MST)**. The segment tree organizes history into precomputed sub-intervals, so that any range query can be answered by combining a small, fixed number of nodes. The vector commitment then authenticates each node with a single, constant-size proof. Together, they enable range-query proofs whose size does not grow with the queried history.

Combining segment trees with vector commitments raises two challenges. First, vector commitments have a fixed capacity, limiting how much history any single commitment can capture. Smaran addresses this with a hierarchical architecture: when a commitment reaches capacity, its MST's root is promoted into a parent commitment, allowing the structure to scale to arbitrarily long histories while keeping each commitment within bounds. Second, existing commitment schemes have no native notion of time. Smaran addresses this

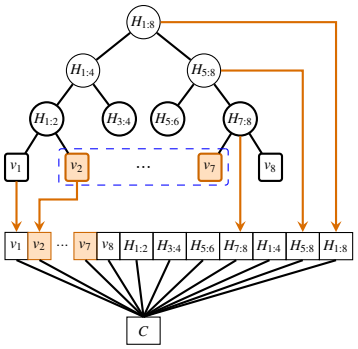


Figure 2: Flattening the MST yields one commitment over all nodes.

by dividing time into discrete epochs, where each epoch corresponds to a published commitment on the ledger. Organizing commitments by epoch enables time-bounded queries such as "all transactions in the past 6 months." This model also aligns naturally with existing distributed-trust applications, which already publish a commitment to the ledger at the end of each epoch [18, 38, 41, 49]. In practice, however, most epochs produce no updates to a given record. Standard schemes handle this by requiring a value at every epoch, effectively copying the previous value into each inactive epoch—an overhead that grows with the number of epochs. Smaran eliminates this redundancy by introducing native versioning into the commitment layer, so that inactive epochs require no space at all.

The rest of this section describes how Smaran realizes these ideas, starting with how it organizes record versions into a structure that supports efficient time-travel queries.

Flattening the MST. Smaran maintains an MST for each record, where leaves represent the record's versions and each internal node is the hash of its two children. On its own, however, this design offers no advantage over a standard Merkle tree: the root commits to all leaves, but a time-travel query still requires a separate proof per version. Since witness generation dominates proof-generation cost, Smaran instead produces a single proof independent of the query range. It achieves this by flattening the MST—arranging all nodes into a contiguous vector. This vector representation allows any query range to be expressed compactly and serves as direct input to the vector commitment scheme. Figure 2 illustrates this for versions $\langle v_1,\ldots,v_8\rangle$: the MST nodes are arranged into a vector, over which a single commitment C is built.

Vector Commitment API. Smaran applies an off-the-shelf vector commitment scheme over the flattened MST using four functions, summarized in Table 1. We discuss the choice of scheme and its capacity constraints in §5.1.

Initialization. When a new record k is created, Smaran initializes it with an empty MST, represented as a vector V of pre-determined size L with all elements set to 0[1], and computes the initial commitment C by calling Commit(V).

[1]We treat 0 as a special null value and ensure that no version v_i encodes to 0.

Function	**Description** (commitments and witnesses are constant-size)	**Cost**
Commit(V)	Takes a vector V of size n; outputs a commitment C	$O(n)$
Update(V,C,i,x)	Updates the i-th element of V to x; outputs updated V' and C'	$O(1)$
Witness(V,B)	Takes a batch B of index-value pairs; outputs a witness w	$O(\lvert B\rvert)$
Verify(C,w,B)	Verifies $V[i]=v_i\ \forall i\in B$ against commitment C and witness w	$O(\lvert B\rvert)$

Table 1: Vector Commitment APIs and expected costs

Updating a Record. Upon receiving a new version v_i of record k, the server computes its hash and inserts it as the i-th leaf of the MST. Parent hashes are then updated recursively: whenever an internal node gains both children, its value is set to $H(\text{left}\|\text{right})$. Throughout this incremental construction, Smaran maintains a *position invariant*: the position of every node and its parent-child relationships remain fixed as new versions are added. This guarantees that every version has a fixed, pre-determined index in the flattened vector V. Once v_i has been inserted, Smaran calls Update(V,C,i,v_i) to produce the updated commitment C'.

5.1 Hierarchical MST

In vector commitment schemes, such as the seminal KZG commitment [30], the maximum supported vector length is fixed during setup by the public parameters (§6.3). This bounds the number of versions a single MST can hold to $\lfloor\frac{L+1}{2}\rfloor$, where L is the maximum vector length (see Supplementary Material for the derivation).

Smaran resolves this limitation through a hierarchical architecture, organizing MSTs across multiple layers.

A *naive hierarchical design* where the leaf nodes of each upper-layer MST are commitments to lower-layer MSTs, increases the proof generation cost. In this design, each lower-layer MST is independently committed, and the upper-layer MST binds these commitments together. While conceptually simple, this design has a significant weakness: answering a time-travel query requires one witness to authenticate the upper-layer commitment, plus one additional witness for each lower-layer commitment touched by the query. The total witness count, therefore, grows with the number of lower-layer MSTs that overlap the query range (see Supplementary Material for detailed discussion of this naive architecture).

In Smaran, upper-layer MSTs treat lower-layer MST roots as leaf nodes–*promoting* the root node (not just commitments) to preserve segment tree structure. This design keeps proof generation and verification to at most two witnesses per layer.

When the current MST is full, Smaran finalizes it by promoting its root into a designated leaf position in the parent MST at the next layer. Subsequent versions are written to a fresh MST at the lower layer; when that MST is exhausted, its root is likewise promoted. This recursive procedure allows Smaran to store arbitrarily many versions while respecting the vector-length bound L at each layer.

Each tree at every layer is itself an MST, so Smaran must commit to its flattened representation. It also stores the commitments of lower-layer trees, as these are needed during

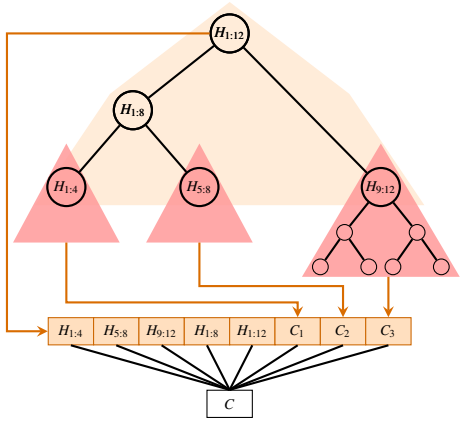


Figure 3: Smaran's hierarchical MST

proof generation. The flattened representation of an upper-layer MST therefore contains: (1) the roots of m lower-layer trees, serving as leaf nodes; (2) the $m-1$ internal nodes of the upper-layer MST; and (3) the commitments of the m lower-layer trees. This limits the number of lower-layer roots an upper-layer MST can hold to $m = \lfloor \frac{L+1}{3} \rfloor$ (see Supplementary Material). Smaran ensures that this multi-layer construction preserves the position invariant. Beyond correctness, the structure supports a total of $n \cdot m^{\gamma-1}$ versions across γ layers; in practice, $\gamma = 4$ is sufficient for any real-world deployment (see Supplementary Material for the derivation).

Figure 3 illustrates our hierarchical construction for $L = 8$, where the upper-layer MST has three leaf nodes ($H_{1:4}$, $H_{5:8}$, and $H_{9:12}$), each for the root node of a lower-layer MST, along with two internal nodes and three lower-layer commitments. For a query over $\langle v_1, \ldots, v_{12} \rangle$, a single witness authenticating the upper-layer commitment C suffices, as the upper-layer root directly commits to all versions.

5.2 Public Verifiability

Smaran maintains one MST per record, producing a commitment that is updated whenever a new version is created. Publishing these commitments to a publicly accessible ledger is a core requirement for public verifiability. Since applications employing Smaran may serve billions of records, publishing each commitment individually to the ledger is impractical. Thus, Smaran follows the established practices of applications with an untrusted query server: aggregating all per-record commitments in a Merkle Patricia Trie (MPT) [22], mapping each record identifier k to its current MST commitment and publishing only the MPT root to the ledger.

Unlike Merkle trees, MPTs commit to key-value mappings via trie structure: keys represented as bytes dictate paths, with shared prefixes stored once to eliminate duplication. Like a Merkle tree, the MPT root changes whenever any record's MST commitment is updated. The witness for a record k has size $O(|k|)$, where $|k|$ denotes the length of the record identifier. The next section describes how this MPT witness is composed with the MST witness to verify the correctness and completeness of time-travel query results.

5.3 Query Procedures

Upon receiving a time-travel query, the server generates a proof and returns it to the client, which then verifies its correctness and completeness. Smaran is designed to optimize both operations through its MST structure. Specifically, we describe proof generation and verification for a query over version range $[i:j]$ of record k, where C denotes the MST commitment for k and C_M denotes the MPT root.

Proof Generation. To generate a proof for a range query over $[i:j]$, Smaran must produce: (1) witnesses that prove C commits the queried versions, and (2) a witness w_M that proves C_M commits C.

Smaran generates MST witnesses by starting at the root of the hierarchical structure and performing a depth-first search to locate the nodes that cover $[i,j]$. Because each tree in the hierarchy is an MST (i.e., a segment tree), this interval can always be expressed as a union of at most $2\gamma-1$ disjoint segments, each answered by a range query on a single MST. In particular, segment-tree properties guarantee that at each layer we touch at most two MSTs, and that no two of these trees lie on the same root-to-leaf path. Intuitively, the query range is a single contiguous block of versions; at each layer, its central portion is covered by internal nodes of that layer's MST, while any remaining versions lie strictly to the left or right. Each such sub-range begins at the root of an MST in the next layer, and the same argument applies recursively.

Once Smaran has identified the set $\mathcal{T}$ of MSTs touched by the query, it constructs a witness bottom-up for each $t \in \mathcal{T}$. For each t, Smaran assembles a witness batch B_t of all nodes covering the sub-range that t answers and calls Witness(V, B_t) on t's flattened vector V. To facilitate verification, Smaran also sends the commitment C_t for each t to the client. To authenticate tree t, its commitment C_t is included in the witness batch $B_{t'}$ of the upper-layer tree t' that covers t. Together, these steps amount to a single segment-tree traversal per $t \in \mathcal{T}$; this yields a proof generation cost of $O(\log n)$.

Example. Consider a query for versions $[4:9]$ of record k over the hierarchical MST in Figure 3. The query touches three lower-layer MSTs: the last version in the MST for $[1:4]$, all versions in the MST for $[5:8]$, and the first version in the MST for $[9:12]$. Since $[5:8]$ is a leaf node of the upper-layer MST, Smaran creates a witness for C_2 directly. For the left and right parts, it traverses the corresponding lower-layer MSTs down to the leaf level, creating a witness for each. Finally, Smaran generates an MPT witness showing that C_M commits record k's MST commitment C.

Proof Verification. To verify the response for a range query over $[i:j]$, the client first reconstructs the MST nodes that the received witnesses are expected to authenticate, reading each returned version in the range. The client also fetches the most recent MPT root C_M and uses w_M to confirm that C_M commits record k's MST commitment C. Finally, for each

$\langle C_t, w_t \rangle$ pair, the client calls Verify(C_t, w_t, B_t), where B_t is the subset of reconstructed nodes belonging to $t \in \mathcal{T}$.

The cost of each step follows directly from its structure. The MST reconstruction cost is linear ($O(j-i)$), since the client must read every version in the range. MPT verification costs $O(|k|)$, and verifying all MST witnesses costs $O(\log n)$, yielding a total verification cost of $O(j-i+|k|+\log n)$.

Example. Consider the same query for versions $[4:9]$ of record k over the hierarchical MST in Figure 3. The client receives the versions $\langle v_4, v_5, ..., v_9 \rangle$, the MPT witness confirming that C_M commits C, and the MST commitment-witness pairs for the same three pieces identified during proof generation: the MST for $[1:4]$, the MST for $[5:8]$ authenticated at the upper layer, and the MST for $[9:12]$. From the returned versions, the client reconstructs the MST nodes for these three pieces and verifies each witness against its corresponding commitment. If all checks succeed, the client accepts that the versions in $[4:9]$ are authentic and complete.

6 Implementation

We implement Smaran in Go in 8.9k lines of code. For hashing, we use Keccak256, a collision-resistant hash function, and for generating vector commitments, we use Gnark-crypto KZG implementation [13], which follows the popular KZG commitment scheme [30].

Ingestion. Smaran is designed so that maintaining verifiable history does not degrade the throughput or latency of the underlying system. To achieve this, Smaran adopts a multi-threaded pipelined architecture. At a high level, Smaran initializes a pool of workers (goroutines) that are responsible for incrementally maintaining a hierarchical MST for each record. A dedicated *reader worker* continuously scans the stream of valid transactions–that is, those that have already passed consensus–and decomposes each transaction into a set of primitive operations. For example, in a distributed banking setting, a transfer where Alice sends \$10 to Bob is represented as two operations: decrement Alice's balance by \$10 and increment Bob's balance by \$10. Each operation is then individually hashed and assigned to one of several per-worker queues, effectively partitioning the operation stream into buckets. Every such queue is owned by a dedicated *processor worker*, which consumes operations from its queue and applies them to the corresponding MSTs.

Querying. To ensure that query processing does not interfere with transaction ingestion, Smaran runs queries on a separate pool of dedicated workers. When a client issues a query, an available query worker picks it up and identifies the target record k along with the requested range $[i:j]$. The worker then fetches the required MST nodes from memory or disk, constructs the corresponding cryptographic witnesses, and returns both the query result and its proof to the client.

Smaran targets a broad class of applications where the query server is untrusted, so the design must adapt to different deployment contexts. In this paper, we instantiate it in two concrete settings: (i) Key Transparency, and (ii) Decentralized Ledger. Each of these applications imposes its own constraints, which in turn require specific design choices to efficiently support authenticated time-travel queries, which we describe next.

6.1 Key Transparency

As described in Section 2, key transparency systems support monitoring queries: prove that the key of client k has α versions, which requires proving the integrity of versions 1 through α and creating a non-membership proof for version $\alpha+1$. Smaran proves the integrity of versions 1 through α through a range query over the indices $[1:\alpha]$. It proves the non-membership of version $\alpha+1$ (or that α is the most recent version), by slightly modifying the Merkle Patricia Trie (MPT). For each client k, the stored value is no longer just the commitment C to its MST; instead, Smaran stores $H(C \,||\, v_\alpha \,||\, \alpha)$, where v_α is the latest version of client k's key and α is the version number. This is sufficient to convince the client that α is the most recent version of record k because the client can verify the returned latest key and version against the MPT proof.

Optimization. Smaran allows clients to be offline for an arbitrary number of epochs. A client can reuse the results of past monitoring queries and needs to download only the versions that were added after its last check, rather than re-fetching the entire history. Moreover, if the client caches the MST nodes returned by those queries, it only needs to store $O(log\ n)$ MST nodes instead of all n key versions.

6.2 Decentralized Ledger

Versioning and Redundancy. In decentralized systems, time-travel queries typically ask for a record's state across a range of epochs (time steps or blocks). For instance, a client preparing an annual tax report might query how their account balance evolved over the past financial year. To answer such queries correctly, the system must link each version of a record to its corresponding epoch. In practice, however, record updates follow a power-law distribution–most records change infrequently and do not have a new version at every step. Existing commitment schemes struggle with this sparsity: they assume every epoch contributes a value. A simple workaround is to copy the last known version into every empty epoch, ensuring that each epoch has some value to commit. Yet this naïve approach inflates redundancy.

Smaran tackles this redundancy by separating the historical versions from the most recent one. If a record k has n versions, Smaran stores versions v_1 to v_{n-1} in the MST along with the start and the end epochs in which these versions appeared. For example, if the leaf nodes of the MST are $\langle v_1,1,3 \rangle$, $\langle v_2,4,8 \rangle$,..., $\langle v_5,20,25 \rangle$, it means that k had version v_1 till end of epoch 3 and version v_5 from epoch 20 to 25, while the current version, v_6, appeared in epoch 26. For record k, the Merkle Patricia Trie (MPT) includes a corresponding value $H(C \,||\, v_6 \,||\, 6 \,||\, 26)$. When a new version is introduced in some future epoch (say, epoch 30) Smaran adds $\langle v_6,26,29 \rangle$

Operation	Latency (ns)
Keccak256 Hash (64 byte input)	368
Hash to BLS12-381 Curve Point	82 290
BLS12-381 Pairing	775 437

Table 2: Latency of cryptographic operations in Smaran

to MST, rebuilds the MST commitment C' and updates the MPT entry to $H(C' \,||\, v_7 \,||\, 7 \,||\, 30)$.

Archival Storage. As the total size of Smaran's MSTs grows over time, the system eventually relies on on-disk storage to maintain historical versions. This introduces a tension between storage efficiency and query performance. One extreme is to store only the leaf data–the individual record versions. This minimizes storage but forces Smaran to rebuild entire segment trees at query time, yielding query complexity of $O(n)$ per record. At the other extreme, Smaran could persist on every node in each MST, including all internal nodes, which removes any need for recomputation but doubles storage requirements. Smaran adopts a balanced approach. Instead of storing all internal nodes, it persists only the leaves and root commitments of each MST.

Sharding. Smaran further exploits natural parallelism in its design: manipulations of MSTs for different records are embarrassingly parallel. To harness this property, Smaran organizes records into logical shards, distributing them evenly across available processing threads. Each worker can then perform tree construction and updates independently without coordination overhead. While MPT operations are inherently sequential due to data dependencies, Smaran's design avoids performance loss by parallelizing MST updates instead. Since the elliptic-curve operations used for generating commitments are substantially slower than hash computations, we allow updates to MSTs and MPT to run in parallel.

6.3 KZG Commitment Details.

The KZG commitment scheme requires public setup parameters–known as the Structured Reference String (SRS)–generated once and published globally. Commitment generation, witness generation, and verification rely on the SRS. The size of the SRS dictates the maximum vector size L. Following practical deployments of KZG Commitments [21], we choose an SRS of size 4096.

KZG commitments use pairing-based cryptography [30] using an underlying pairing-friendly elliptic curve. We use the curve BLS12-381 [28] for this purpose. The commitment and the witnesses are simply points on the curve and are of constant size at 48 bytes. To verify a witness, the client performs a pairing check operation. The cost of performing these operations is summarized in Table 2.

With our current setup of KZG commitments, even with $\gamma = 4$, Smaran can handle more than five trillion versions for a record! We optimize our incremental commitment generation using the Barycentric method [5], so that commitment generation only ever requires $O(\gamma)$ elliptic curve operations. As is standard, Smaran uses Inverse FFT [14] to compute polynomial interpolations during witness computation. During verification, while Smaran may use twice as many hashing operations as the queried version range, it performs at most $2\gamma+1$ pairing checks.

We provide a detailed overview of the KZG commitment scheme in the Supplementary Material.

7 Evaluation

Our evaluation aims to answer the following three questions:

1. Does Smaran enable efficient authenticated time-travel queries (i.e., properties succinctness and efficiency, §3)?
2. What overhead does Smaran impose on the representative applications (i.e., properties robustness and generality)?
3. How effective are Smaran's design considerations?

To answer these questions rigorously, we integrate Smaran into two representative applications: key transparency and decentralized ledgers, each with distinct baselines and workloads.

Experimental Setup. We deploy two CloudLab (Clemson) machines: a server on an r6615 node (AMD EPYC 9354P, 32 cores, 192 GiB RAM, 1.6 TB NVMe SSD) and a client on a c6420 node (Intel Xeon Gold 6142m, 16 cores). Both machines run Ubuntu 22.04. Each data point reports the average of *three* runs. We discard the first and last 30 seconds of each experiment to exclude warm-up and cool-down effects. Our experiments focus on two key metrics: (1) *throughput* (operations per second) and (2) *latency* (client request to response time).

7.1 Key Transparency

First, we measure the effectiveness of Smaran as a key transparency system.

Baselines. We compare Smaran against two popular key transparency systems: CONIKS [41] and OPTIKS [35].

CONIKS. It operates epoch-by-epoch: it publishes a signed MPT [2] root per epoch representing the current directory state. Monitoring a key after T epochs requires verifying one proof per epoch, yielding T proofs total–with proof size and verification cost scaling linearly with T. We deploy the official Go implementation of CONIKS [8].

OPTIKS. It organizes key transparency around version history: it stores all versions of a user's key together and proves monitoring queries from the latest root of MPT. Clients verify all past versions exist and no next version follows–yielding one proof per version (v total), with proof size and verification cost scaling linearly with v. Since no public implementation is available, we implement OPTIKS following the design described in the paper.

[2] OPTIKS uses a Merkle Prefix Tree, which is a primitive version of the Merkle Patricia Trie that we adopt for Smaran. They have the same asymptotic operational complexities.

System	Proof gen.	Proof size	Verify
CONIKS	$O(T\log n)$	$O(T\log n)$	$O(T\log n)$
OPTIKS	$O(v\log n)$	$O(v\log n)$	$O(v\log n)$
Smaran	$O(\log v+\log n)$	$O(\gamma)$	$O(v+\log n)$

Table 3: Asymptotic cost of key transparency monitoring queries.

Complexity Analysis. As described in §4, key transparency systems are interested in monitoring queries (detecting malicious key injections).

In Table 3, we summarize the asymptotic costs of proof generation, proof size, and verification. For monitoring queries, CONIKS returns signed commitment chains with Merkle proofs per epoch, giving $O(T\log n)$ proof generation, proof size, and verification cost, where T is the number of epochs since the last monitoring operation and n is the number of users. OPTIKS instead proves membership across all key versions using the latest commitment, giving $O(v\log n)$ proof generation, proof size, and verification cost, where v is the number of key versions. In contrast, Smaran achieves $O(\log v+\log n)$ proof generation cost, $O(\gamma)$ proof size, and $O(v+\log n)$ verification cost.

Workloads. We evaluate these protocols by generating a realistic key transparency workload with 10K users, where each key supports up to 2047 versions. For CONIKS, which tracks epochs rather than versions (with ≤ 1 key update per epoch), we simulate up to 2047 epochs.

Monitoring Query Performance. We first evaluate monitoring query performance using 10 concurrent clients, each of which randomly selects a user for continuous monitoring. All data fits in memory to isolate protocol overheads. For Smaran, we configure $\gamma=1$ layer (single MST per key), sufficient for our 2047-version maximum. Figure 4 compares Smaran's performance against baselines.

Smaran's throughput declines gradually as the number of versions increases, decreasing from 539 ops/s at 2 versions to 239 ops/s at 2047 versions. Its latency likewise increases gradually, from 18.5 ms to 41.6 ms over the same range. This is because Smaran only needs to authenticate the MST nodes that cover the queried history, which requires touching at most $2\times log(2048)=22$ internal nodes (a single internal node can cover a large segment of the queried range). In contrast, both OPTIKS and CONIKS exhibit a much steeper, approximately linear decline in throughput and increase in latency because they must generate proof material proportional to the number of returned versions or epochs. Consequently, at 2047 versions, Smaran reduces end-to-end latency by 99.1% relative to CONIKS and by 82.5% relative to OPTIKS, while delivering 113× higher throughput than CONIKS and 5.7× higher throughput than OPTIKS.

Payload. We also measure the amount of payload each protocol has to send to the client as a response to the monitoring query. For all protocols, there is an increase in payload size with an increase in the number of versions: each protocol must return to the client the key versions and the witnesses. Smaran has the smallest payload because it is proportional to the number of layers in the hierarchical MST. When $\gamma=1$, the MST witness consists of a single 48-byte BLS12-381 curve point. CONIKS and OPTIKS, by contrast, require one witness per version/epoch. At 2047 versions, Smaran's payload size is 160.7 KiB, which is 97.0% lower than CONIKS and 95.2% lower than OPTIKS.

Key Update Performance. Next, we evaluate the cost of maintaining continuously growing key-version histories. For this experiment, we initialize a directory of 10K to 1M users, with each user starting with one key version, and then measure insertion throughput under realistic churn. Each epoch applies 2^{14} updates (∼1.6% of users issue update requests in the 1M-user setting), executed in batches to reflect production key-rotation patterns.

Figure 5 shows key-update throughput as directory size scales from 10K to 1M users. Smaran consistently outperforms CONIKS and trails OPTIKS slightly across all scales. At 10K users, Smaran yields about 130× higher throughput than CONIKS, while remaining 27.0% lower than OPTIKS. At 1M users, Smaran delivers about 115.6× higher throughput than CONIKS and closes the gap with OPTIKS to just 14.8%. This trend reflects the underlying update costs: Smaran is slower than OPTIKS because BLS12-381 polynomial-commitment operations are more expensive than hash-based Merkle updates, but the gap remains small because OPTIKS must still traverse multiple levels of its authenticated structure on every key change. CONIKS, by contrast, suffers higher update costs: it authenticates the entire directory per epoch. Each key update requires recomputing the MPT path to root and publishing a new signed root–limiting throughput to hundreds of operations per second in comparison to Smaran and OPTIKS.

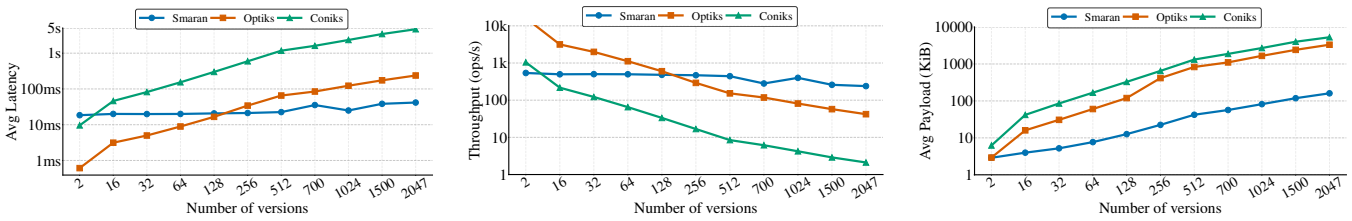
7.2 Decentralized Ledger

Next, we measure the effectiveness of Smaran in a Decentralized ledger that supports authenticated time-travel queries.

Baselines. We compare Smaran against three popular schemes: Merkle Patricia Tries [22], Verkle trees [9], and Cauchyproofs [39].

MPT. As described in §5.2, MPTs enable public verifiability. However, many production systems use MPTs only for authenticated proofs, without additional mechanisms for time-travel queries [18]. Consequently, it answers range queries with one Merkle proof per version, causing both proof size and verification cost to grow linearly with range length.

Verkle Trees. They organize state in a high-branching factor trie where each internal node commits to its children using vector commitments. This yields smaller proofs than MPTs for point queries, but answering a range query still requires generating a separate proof per version, with each involving expensive elliptic-curve operations.



(a) Latency. *(b) Throughput.* *(c) Payload size.*

Figure 4: Performance of key transparency systems on monitoring queries versus number of key versions.

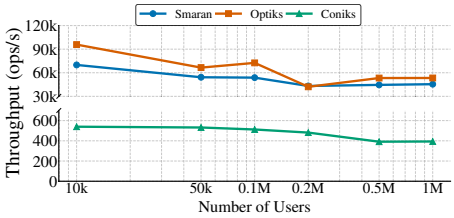


Figure 5: Key update throughput as the number of users increases.

Cauchyproofs. It represents the entire state with a single vector commitment and maintains a witness for each account that proves the latest state. To answer a time-travel query, the server starts from the latest state and walks backward through the updates in the queried interval to recover the witnesses for the requested past versions. Cauchyproofs [39] is the state-of-the-art solution for answering authenticated point queries, and thus we test its design on time-travel queries.

For testing MPT and Verkle trees, we use the open-source Go implementations from Ethereum (go-ethereum [19] and go-verkle,[3] respectively). For Cauchyproofs, we build on the authors' anonymously released implementation, which includes only the core vector-commitment algorithms. We implement the additional system components needed to serve time-travel queries.

Workloads. For our experiments, we deploy a production decentralized ledger by setting up an Ethereum node. Additionally, we download *one full year* of Ethereum blocks (from 1st January 2024 to 31st December 2024), totaling 2.6 million blocks, all of which can be publicly verified. The total number of unique accounts in these blocks is 60 million. The distribution of updates received by these accounts follows a strict power law: the most active account had 2.6 million versions (approximately one per block), whereas even the 100^{th} most active account had only 534k versions.

Due to the large size of the data, we can no longer store all the data in memory. Thus, we employ PebbleDB [12] for on-disk storage. PebbleDB is an LSM tree-based KV store. Smaran uses it to store the historical and current states of each account, as well as the KZG commitments and root hashes of each MST. Additionally, we logically shard accounts to enable parallel processing across multiple accounts. Thus, we deploy *three* PebbleDB instances per shard: (1) to store the latest MST (where the next version will be added) of each layer for each record, (2) to store KZG commitments, root hash of each MST, and the latest versions, and (3) to store the historical versions. Merkle and Verkle schemes also employ PebbleDB to store the tree nodes (one tree state snapshot per block).

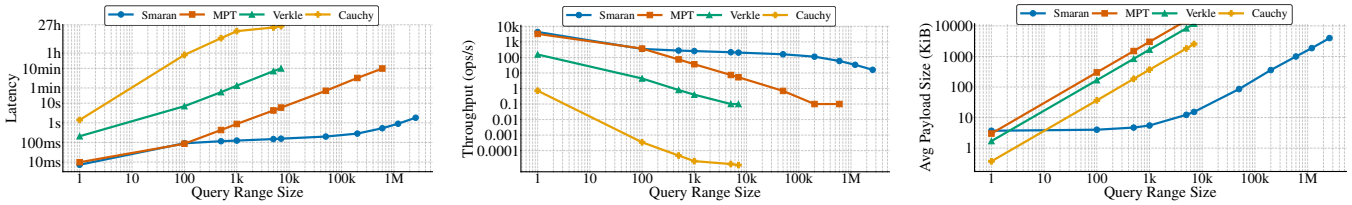
Query Performance. We first evaluate query performance with 32 concurrent clients, each randomly selecting an account from a weighted distribution over the 1,000 most frequently updated accounts and requesting a proof for that account's versions in the last n epochs; we increase n from 1 to 2.6M. Specifically, we issue time-travel queries mimicking real-world account statement periods: 4 hours, 1 day ($\sim$ 7K epochs), 1 week, 1 month, 3 months, 6 months, and 1 year (Ethereum epoch = 12 seconds). For Smaran, we configure MST to $\gamma=4$ layers. Figure 6 compares Smaran's performance against baselines on increasing the query range.

For a query range of size 1, Smaran yields a throughput of $\sim$4,352 ops/s, which is 1.3×, 28×, and 6217× higher than Merkle, Verkle, and Cauchy. Throughput declines across all protocols as query range increases, but baselines degrade much faster due to their linear proof-generation costs. For MPT, each range query of size n requires $O(n)$ database lookups to fetch tree nodes and n Merkle witness generations. Verkle also generates per-epoch witnesses but using expensive elliptic-curve operations, making it slower than hash-based alternatives despite smaller individual proofs. Cauchy yields negligible throughput due to the high cost of reconstructing the required past witnesses from the latest maintained state. In contrast, Smaran needs only $O(\gamma)$ MST lookups and $O(\gamma)$ cryptographic witness generation plus one MPT proof, regardless of the query range. With $\gamma=4$, this accounts for at most 7 MST witnesses per query.

Consequently, Merkle and Verkle observe a 90.19%, and 90.91% drop in throughput from range size of 100 to 1k, and 85.43% and 75% drop in throughput from range size of 1k to 7k. In comparison, for the same query ranges, Smaran only observes a drop of 27.05% and 20%, respectively.

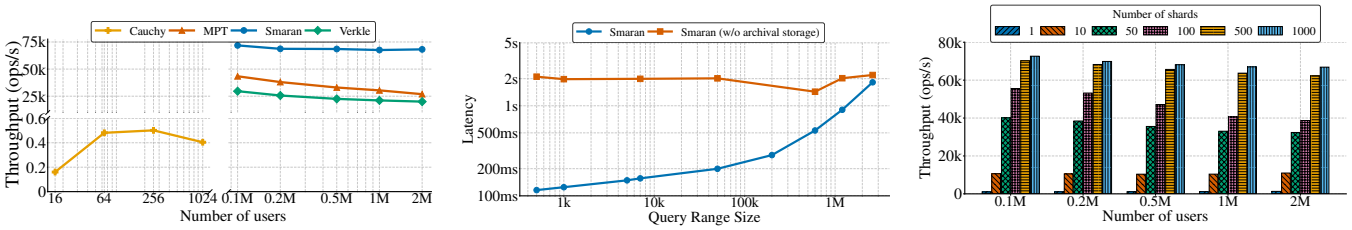
Moreover, running these baselines is computationally prohibitive: generating Cauchyproofs for a query range of size 100 took us $\sim$48 minutes and over $\sim$20 hours at range 5k. Similarly, Verkle becomes impractical beyond the query

[3] Go-verkle is based on Pedersen commitments [20], another elliptic curve commitment scheme.



(a) Latency. *(b) Throughput.* *(c) Response payload size.*

Figure 6: Performance of different commitment schemes on time-travel queries in a decentralized ledger as query range size increases.



(a) Commitment Generation Throughput. *(b) Impact of archival storage.* *(c) Impact of sharding.*

Figure 7: Design considerations for deploying Smaran in a decentralized ledger setting.

range of size 7k ($>$ 9min per query) and Merkle beyond range 600k ($>$ 9 min per query). Smaran scales effectively even at 600k versions, delivering 99.94% lower latency and 589$\times$ higher throughput than Merkle trees. Furthermore, it maintains query performance for ranges as large as 2.6M.

The verification cost at the client side also grows linearly for all but Smaran. However, the cost to verify just $O(\gamma)$ witness makes it more expensive than MPTs. For example, at a query range of 600k, verifying a query for Smaran takes only 204ms, versus Merkle, which takes 37.8s.

Payload. As each protocol needs to send the proof back to the client, we measure the payload size. For all protocols, the payload size increases with an increase in query range. However, baselines degrade faster: MPT, Verkle, and Cauchy's payload increases by 7017.57$\times$, 6929.18$\times$ and 6374.5$\times$ from range 1 to range 7k, respectively. In contrast, for the same range, Smaran's payload size only increases by 4.14$\times$.

Commitment Generation Performance. Next, we evaluate commitment update throughput on increasing the number of accounts. Figure 7a illustrates ingestion performance on gradually growing the system up to 2M accounts (the accounts are selected in decreasing order of update frequency).

Both MPT and Verkle exhibit declining throughput as accounts scale: their global trie depth grows logarithmically, inflating per-update path costs. Smaran avoids this entirely–each account maintains an independent MST, so per-account update cost depends only on individual MST depth, not on the total number of accounts. Cauchy again has an extremely low throughput, remaining below 0.6 ops/s, because each state update requires both commitment generation and witness recomputation for every tracked account under the new state.

In summary, Smaran sustains its throughput across all sizes and yields up to 2.54$\times$ and 3.41$\times$ higher throughput than MPT and Verkle for 2M users. Moreover, while the drop in throughput for Merkle and Verkle is 38.1% and 32.5% on increasing the number of accounts from 0.1M to 2M, respectively, Smaran observes a drop of only 5%.

7.3 Design Considerations

Archival Storage. Smaran must persist at minimum two components: version hashes at the lowest layer (the actual record data) and commitments for each MST. All other MST nodes can, in principle, be recomputed from these. However, Figure 7b shows that also storing each MST's root hash significantly improves query performance. Without stored roots, every query must reconstruct flattened MST vectors for witness generation. Commitments alone cannot suffice–the polynomial must interpolate over all vector elements, including MST nodes. Smaran must therefore recompute these nodes recursively from leaves: upper-layer leaves are lower-layer MST roots, cascading down to version hashes. This makes query cost largely independent of range size: latency sits at $\sim$1.9s across all ranges.

Storing each MST root breaks this cascade. Smaran fetches only the query-relevant MST leaves–version hashes at the base layer, stored roots at upper layers–then locally recomputes internal nodes via hashing. Reconstruction is confined to touched MSTs, requiring just $O(\gamma)$ database lookups total. At range 1k, latency drops from $\sim$1.9s to $\sim$19 ms (a 99$\times$ reduction). At range 7k the gap is 75$\times$ ($\sim$25ms vs $\sim$1.9s). As the range grows, the two strategies converge because the dominant cost shifts to fetching and returning the versions themselves: at range 2.6M, both reach $\sim$1.2-1.6 s. For 2.6M blocks, storing MST roots adds $\sim$2GB (less than 2% increase in total storage), in exchange for up to two orders of magnitude lower latency at smaller ranges.

Sharding. A core design choice in Smaran was the logical sharding of account data for parallel access. Figure 7c shows ingestion throughput scaling from 1 to 1k shards across 100k–2M accounts. Single-shard throughput plateaus at $\sim$ 1.1K ops/s due to serialized MST updates; 10 shards deliver $\sim 10\times$ speedup ($\sim$ 10.5K ops/s), confirming embarrassingly parallel per-record updates. Scaling continues: 50 shards hit $\sim 33-40$K ops/s, 500 shards $\sim 62-70$K ops/s. Beyond 500, gains diminish as the 32-core server saturates; 1000 shards increase throughput by only 7.4% for 2M users.

Large shard counts also stabilize throughput as accounts grow: at 1k shards, it drops just 8% (72.6k $\rightarrow$ 66.9k ops/s) from 100k to 2M accounts. Fewer shards amplify degradation–at 20 shards, over the same range, there is a 19% drop–as each worker handles more records. These results validate our choice of 1k shards for query experiments.

8 Related Work

Authenticated data structures. Authenticated data structures such as Merkle trees [42] and Merkle Patricia Tries [22] are the standard way to authenticate a committed state. They are widely used because they make point queries simple: the server returns the requested value together with the short path that lets the client recompute the committed root. This works well when a query asks for one value at one state. When the goal is to answer time-travel range queries, however, the same approach becomes much less attractive, since the server must effectively provide proof material for each returned version. As a result, both proof size and verification work grow with the length of the queried history.

Cryptographic accumulators. Cryptographic accumulators [6, 29, 32, 53] offer another way to bind a set of values into a single commitment, since each produces a constant-size binding over an arbitrary set of values. However, all three involve significantly heavier algebraic machinery than hash-based constructions. Smaran uses Merkle hashing at each segment-tree node instead, which is simpler and more efficient in practice.

Succinct proofs from vector commitments. A different line of work uses vector commitments, which let a server commit to many values and later prove selected positions succinctly. Systems such as PointProofs [23], HydraProofs [45], and NTTProofs [25] make these proofs smaller, easier to combine, or cheaper to update. While these improvements are appealing when state changes frequently, all three schemes are built around individual positions, so a long history query is still handled as many separate proofs.

Cauchyproofs [39] is the most closely related prior work. Unlike the schemes above, it explicitly considers the cost of maintaining proofs as updates arrive, making it well-suited for dynamic settings. Its history query algorithm, however, produces a separate proof for each version in the queried range. Smaran departs from this in one key way: rather than assembling interval query answers from many per-version proofs, it answers interval queries with a single constant-size proof by construction, via the MST structure.

Transparency logs. Transparency logs such as Certificate Transparency [34], append-only authenticated dictionaries [52], and Sigstore/Rekor [49] use authenticated append-only structures to provide publicly verifiable records of events. These systems support inclusion, consistency, and efficient lookup proofs, but they are designed for log entries and key lookups rather than authenticated interval queries over the version history of an individual record.

Authenticated histories in deployed systems. Several works demonstrate the need for authenticated histories. Key transparency systems such as CONIKS [41], SEEMless [11], Parakeet [40], OPTIKS [35], VeRSA [53], and Merkle Square [24] let users verify lookups, track changes to their own entries, and detect equivocation or other server misbehavior over time. VerSum [54] permits verifiable computation over large public logs, while Vault [36] enables authenticated bootstrapping and state verification for blockchains. Although these systems target transparency and auditability, they do not present efficient mechanisms for performing authenticated time-travel queries over a record's history.

Permissioned ledger systems such as IA-CCF [47] provide publicly verifiable receipts, enabling auditors to detect inconsistencies and blame misbehaving parties. However, they focus on execution accountability rather than authenticated time-travel queries over record histories. Smaran is complementary to this line of work: it addresses a problem that arises across all of these systems, namely, how to answer historical queries with proofs whose size does not grow with the number of versions.

9 Conclusion

In this paper, we introduced Smaran, a system designed to facilitate efficient, authenticated time-travel queries. Unlike existing solutions that impose high computational and communication costs on provers and verifiers, Smaran offered an authenticated data structure that allowed servers to answer queries with a constant number of fixed-size, publicly verifiable proofs, regardless of the range length. Smaran achieved this by uniquely integrating segment trees with vector commitments. Furthermore, Smaran sidestepped historical scalability challenges through a hierarchical architecture and eliminated data redundancy by introducing versioning to its vector commitments. Additionally, we illustrated Smaran's practical utility by deploying it in two representative applications: key transparency and decentralized ledgers.

References

[1] Advanced Micro Devices, Inc. Amd secure encrypted virtualization (SEV), 2026. AMD developer documentation.

[2] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. Succinct: Enabling queries on compressed data. In *12th USENIX Symposium on Networked Systems Design and*

Implementation (NSDI 15), pages 337–350. USENIX Association, 2015.

[3] Alchemy. Alchemy. https://www.alchemy.com/, 2024.

[4] Aris Anagnostopoulos, Michael T. Goodrich, and Roberto Tamassia. Persistent authenticated dictionaries and their applications. In *Information Security Conference (ISC)*, volume 2200 of *Lecture Notes in Computer Science*, pages 379–393. Springer, 2001.

[5] Jean-Paul Berrut and Lloyd N. Trefethen. Barycentric lagrange interpolation. *SIAM Review*, 46(3):501–517, 2004.

[6] Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In *Advances in Cryptology – CRYPTO 2019*, pages 561–586. Springer, 2019.

[7] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

[8] Arlo Breault, Ismail Khoffi, Marcela Melara, and Quoc Huy Vu. coniks-go: A coniks implementation in golang. https://github.com/coniks-sys/coniks-go, 2016.

[9] Vitalik Buterin. Verkle trees. https://vitalik.ca/general/2021/06/18/verkle.html, 2021. Ethereum research blog post.

[10] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, pages 173–186, 1999.

[11] Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. Seemless: Secure end-to-end encrypted messaging with less trust. In *CCS*, 2019.

[12] CockroachDB, Inc. Pebble: A rocksdb-inspired key-value store. https://github.com/cockroachdb/pebble, 2020. Accessed: 2026.

[13] Consensys. gnark-crypto. https://github.com/consensys/gnark-crypto, 2026. Accessed: 2026.

[14] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[15] DBOS. Instant database time travel queries. https://www.dbos.dev/blog/database-time-travel, 2024. Accessed: 2026-03-07.

[16] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *More Geometric Data Structures*, pages 209–231. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[17] DoltHub. Querying history — dolt documentation. https://docs.dolthub.com/sql-reference/version-control/querying-history, 2025. Accessed: 2026-03-07.

[18] Ethereum Foundation. Ethereum whitepaper. https://ethereum.org/en/whitepaper/, 2014. Accessed on 09/13/2024.

[19] Ethereum Foundation. Go ethereum (geth): Official go implementation of the ethereum protocol. https://github.com/ethereum/go-ethereum, 2015. Accessed: 2026.

[20] Ethereum Foundation. go-verkle: Verkle tree implementation in go. https://github.com/ethereum/go-verkle, 2021. Accessed: 2026.

[21] Ethereum Foundation. Eip-4844: Shard blob transactions (proto-danksharding). https://eips.ethereum.org/EIPS/eip-4844, 2023. Accessed: 2026.

[22] Ethereum Foundation. Merkle patricia trie. https://ethereum.org/en/developers/docs/data-structures-and-encoding/patricia-merkle-trie/, 2023. Accessed: 2026.

[23] Sergey Gorbunov, Leonid Reyzin, Hoeteck Wee, and Zhenfei Zhang. Pointproofs: Aggregating proofs for multiple vector commitments. Cryptology ePrint Archive, Paper 2020/419, 2020.

[24] Yuncong Hu, Kian Hooshmand, Harika Kalidhindi, Seung Jin Yang, and Raluca Ada Popa. Merkle 2: A low-latency transparency log system. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 285–303. IEEE, 2021.

[25] Lijuan Huo, Libing Wu, Enshu Wang, Jinfei Liu, Chunshuo Li, Zemei Liu, and Zhuangzhuang Zhang. Nttproofs: A maintainable and aggregatable vector commitment with fast openings and updates. *IEEE Transactions on Information Forensics and Security*, 20:2778–2792, 2025.

[26] Intel Corporation. Intel software guard extensions (intel sgx), 2013. Intel technical overview.

[27] Intel Corporation. Intel trust domain extensions (TDX) white paper, 2022. Intel white paper.

[28] IRTF CFRG. Pairing-friendly curves. https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-pairing-friendly-curves-05, 2020. Includes BLS12-381 and optimal Ate pairing details.

[29] Ioanna Karantaidou and Foteini Baldimtsi. Efficient constructions of pairing-based accumulators. Cryptology ePrint Archive, Paper 2021/638, 2021.

[30] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *International conference on the theory and application of cryptology and information security*, pages 177–194. Springer, 2010.

[31] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC, Boca Raton, revised 3rd edition, 2025.

[32] Victor Y. Kemmoe and Anna Lysyanskaya. Rsa-based dynamic accumulator without hashing into primes. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*, CCS '24. ACM, 2024.

[33] Mysten Labs. The sui smart contracts platform. Technical report, Mysten Labs, 2022.

[34] Adam Langley, Emilia Kasper, and Ben Laurie. Certificate transparency. *Internet Engineering Task Force*, 2013. https://tools.ietf.org/html/rfc6962.

[35] Julia Len, Melissa Chase, Esha Ghosh, Kim Laine, and Radames Cruz Moreno. {OPTIKS}: An optimized key transparency system. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4355–4372, 2024.

[36] Derek Leung, Adam Suhl, Yossi Gilad, and Nickolai Zeldovich. Vault: Fast bootstrapping for the algorand cryptocurrency. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[37] Benoît Libert and Moti Yung. Concise mercurial vector commitments and independent zero-knowledge sets. In *EUROCRYPT*, pages 499–517. Springer, 2010.

[38] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. Fast and secure global payments with stellar. In *SOSP*, 2019.

[39] Zhongtang Luo, Yanxue Jia, Alejandra Victoria Ospina Gracia, and Aniket Kate. Cauchyproofs: Batch-updatable vector commitment with easy aggregation and application to stateless blockchains. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 1947–1963. IEEE, 2025.

[40] Harjasleen Malvai, Lefteris Kokoris-Kogias, Alberto Sonnino, Esha Ghosh, Ercan Oztürk, Kevin Lewi, and Sean F Lawlor. Parakeet: practical key transparency for end-to-end encrypted messaging. *IACR Cryptol. ePrint Arch.*, 2023:81, 2023.

[41] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. CONIKS: Bringing key transparency to end users. In *USENIX Security*, 2015.

[42] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, pages 369–378, Berlin, Heidelberg, 1987. Springer-Verlag.

[43] Silvio Micali. Algorand, 2016.

[44] Neon. Time travel — neon docs. https://neon.com/docs/guides/time-travel-assist, 2026. Accessed: 2026-03-07.

[45] Christodoulos Pappas, Dimitrios Papadopoulos, and Charalampos Papamanthou. Hydraproofs: Optimally computing all proofs in a vector commitment (with applications to efficient zksnarks over data from multiple users). In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3421–3439, 2025.

[46] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *CRYPTO*, pages 129–140. Springer, 1991.

[47] Alex Shamis, Peter Pietzuch, Burcu Canakci, Miguel Castro, Cédric Fournet, Edward Ashton, Amaury Chamayou, Sylvan Clebsch, Antoine Delignat-Lavaud, Matthew Kerner, Julien Maffre, Olga Vrousgou, Christoph M. Wintersteiger, Manuel Costa, and Mark Russinovich. Ia-ccf: Individual accountability for permissioned ledgers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 467–483. USENIX Association, 2022.

[48] Sigstore. rekor-monitor. https://github.com/sigstore/rekor-monitor, 2026. (Accessed on 02/23/2026).

[49] Sigstore. Rekor: Software supply chain transparency log. https://github.com/sigstore/rekor, 2026. (Accessed on 02/23/2026).

[50] Sigstore. Sigstore. https://www.sigstore.dev, 2026. (Accessed on 02/23/2026).

[51] Snowflake. Understanding & using time travel. https://docs.snowflake.com/en/user-guide/data-time-travel, 2026. Accessed: 2026-03-07.

[52] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. Transparency logs via append-only authenticated dictionaries. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 1299–1316. ACM, 2019.

[53] Nirvan Tyagi, Ben Fisch, Andrew Zitek, Joseph Bonneau, and Stefano Tessaro. Versa: Verifiable registries with efficient client audits from rsa authenticated dictionaries. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2793–2807, 2022.

[54] Jelle van den Hooff, M. Frans Kaashoek, and Nickolai Zeldovich. Versum: Verifiable computations over large public logs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[55] John Vilk, Emery D. Berger, James Mickens, and Mark Marron. Mcfly: Time-travel debugging for the web. *arXiv preprint arXiv:1810.11865*, 2018.