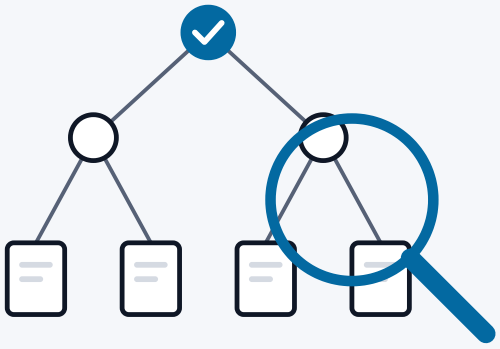
A-SIT Technology

E-Government Innovation Center

29 October 2026 Concept · Transparency · Digital Identity · 28 min read

Verifiable Monitoring for Transparency Logs

Edona Fasllija



Authors Edona Fasllija

Source https://technology2.a-sit.at/en/publications/verifiable-monitoring-for-transparency-logs/

Key findings

- Chronological transparency logs provide inclusion and consistency proofs, but no proof that a content-based query was answered completely.
- A Verifiable Index (log-map-log) maps keys to positions in the input log; an output log records the index revisions.
- With about 16 million leaves, a content-based query takes 858.32 s by log scan, versus 0.047 ms (SMT), 0.100 ms (MPT) or 0.179 ms (IAVL) via the index.
- The index costs build time and storage: about 1.54 GB (SMT), 2.60 GB (MPT), 2.94 GB (IAVL) plus roughly 2.06 GB for the WAL; inclusion proofs barely get faster.
- For Certificate Transparency, a separate index instance per log is proposed; notifications are complemented by periodic heartbeats with proof.

Transparency systems, such as Certificate Transparency and Binary Transparency, let clients verify that a specific object was committed to an append-only public history and that later checkpoints extend earlier ones. What they do not natively provide is a cryptographically complete answer to content-based questions such as “show me all certificates for this domain” or “show me every release for this module”. Users must either scrape and parse the entire log themselves or trust external search/index services that can omit results. Verifiable Indices target this gap and propose a continuously maintained, end-to-end verifiable index layered over an existing append-only log. The index derives content-based keys from log entries, maps them back to positions in the source log, and publishes authenticated index states that can be independently verified. This report presents the Verifiable Index model and architecture, summarizes prototype benchmark results, and discusses its application to Certificate Transparency.

1. Introduction

Transparency systems make security-relevant events publicly auditable by committing them to appendonly logs. The same philosophy appears in certificate issuance, software checksums, build attestations, firmware releases, and other supply-chain settings. Certificate Transparency (CT) is the flagship deployment: certificates and precertificates are recorded in independently operated Merkle-tree logs, and the logs provide compact proofs that an entry is present and that a later tree state extends an earlier one. These properties are highly effective for publication integrity, but they are not a complete monitoring solution. A domain owner usually does not know the log positions of certificates concerning its domains; it needs a content-based query, such as “show me every entry that concerns example.com.” Today this gap is typically bridged by monitors [1]. A monitor collects new entries, parses them, builds its own search index, and exposes search or alerting services. This is operationally effective, but it creates a new trust boundary: a user can verify that a returned certificate is actually logged, yet cannot efficiently verify that the monitor returned every matching certificate. Recent work on third-party CT monitors has shown that incompleteness is not merely theoretical [2].

For chronological logs, however, efficient discovery is not the primitive operation. The log is ordered by time or insertion position. A client that already knows a leaf can request an inclusion proof, and a client that knows two checkpoints can verify append-only growth. Monitoring asks a different question: which entries are relevant to me? This is usually a predicate over content rather than position. A non-indexed log, therefore, gives append-only publication integrity, but not verifiable completeness for predicatebased retrieval. The practical result is a recurring trade-off. A stakeholder can replay and locally index the log, which preserves end-to-end trust but incurs significant storage and processing overhead. Or it can outsource monitoring to a third party, which is operationally attractive but makes completeness a servicelevel claim rather than a cryptographically checkable property.

The core research problem then is not how to replace append-only logs, but how to add verifiable content-based retrieval while preserving the log as the source of truth and keeping monitoring operationally practical. A verifiable index adds an authenticated query layer without replacing the original log. The current transparency-dev prototype describes a “log-map-log” data structure: an Input Log is consumed leaf-by-leaf; a deterministic mapping function converts each entry into one or more lookup keys; a verifiable map associates those keys with pointers back into the Input Log; and an Output Log records successive

revisions of the map so that the map is historically verifiable [3]. This project explores the design in detail, including a write-ahead transaction log (WAL), multiple authenticated map backends, proof-serving APIs, and performance measurements on SumDB-derived data.

2. Background

2.1. Append-only Logs

Append-only chronological logs provide membership and consistency proofs, not a general-purpose searchable database interface. In CT, a client can verify an SCT, an inclusion proof, and a consistency proof between tree heads (STH). A monitor can fetch new STHs, retrieve new entries, classify certificates of interest, and verify that the tree implied by fetched entries matches the new STH or extends a prior STH consistently. However, the protocol does not define a proof that answers a content query, such as "all certificates relevant to example.com." Instead, completeness for that question is achieved operationally by the monitor fetching and parsing every new entry.

That distinction is the key limitation of non-indexed chronological logs. They provide proofs for statements of the form "leaf I is in log n" and "checkpoint n extends checkpoint m," but they do not provide proofs for statements of the form "this returned set is the complete set of all leaves matching predicate P up to prefix n." Verifiable Indices address exactly this point.

In monitoring terms, the cost of this missing property is that content discovery effectively reduces to full log replay. RFC 9162's monitor workflow requires repeated polling for STH changes, fetching all new leaves, checking whether each entry is "of interest," and then either storing the full tree contents or doing enough work to verify consistency against fetched entries and proofs. For high-volume logs this is not merely inconvenient; it is the core scaling bottleneck. A raw chronological log is, therefore, a good publication primitive but a poor query-completeness primitive.

This leads to an important trust trade-off. If users outsource indexing to an ordinary search service, they recover efficiency but lose a formal completeness guarantee. The monitor can omit results accidentally, under load, or maliciously, and the user has no concise cryptographic evidence of omission. The Verifiable Index motivation is precisely to avoid this trade by introducing a "back-of-the-book" index whose answers are themselves covered by cryptographic proofs.

Monitoring as a three-stage pipeline

In practice, few domain owners want to replay every CT log. Third-party monitors, therefore, collect log entries, build databases, expose domain search, and send notifications. The challenge is that CT proofs authenticate the log entry, not the completeness of a monitor's database query. Sun et al. [2] describe third-party monitors as essential CT components and report that popular services do not always return complete certificate sets; their watcher prototype identified multiple implementation or design issues during a 52-day trial.

The security consequences are subtle. A user can receive a certificate from a monitor and verify that it is genuinely logged, but that says nothing about an omitted certificate. Monitoring, therefore, shifts the trust problem from the log operator to the monitor unless the user independently replays the log or the indexing layer itself becomes verifiable.

A useful operational model is to split monitoring into three stages (Figure 1): data collection, indexing, and querying/alerting. Data collection continuously retrieves newly appended records from the transparency logs of interest. Indexing parses those raw records and extracts attributes that become lookup keys. Querying exposes the indexed repository to users, while alerting applies stored queries or subscriptions proactively. This decomposition is valuable because it identifies where trust is introduced.

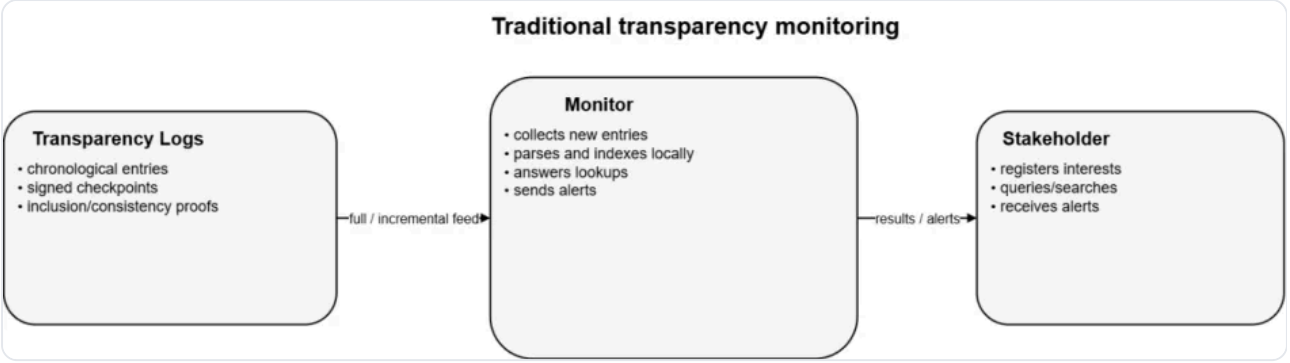


Figure 1: Traditional Monitoring

Data collection must keep pace with every relevant input log and recover correctly after downtime. Indexing must parse entries deterministically and must not silently drop malformed or unexpected records. Querying must return the intended matching set, not merely individually valid entries.

Alerting additionally depends on service and delivery liveness: "no message" can mean "no event" or "the monitor failed". Search is user-initiated. If a query service is unavailable or returns an error, the failure is immediately visible, and the user can retry, switch providers, or inspect the underlying log. Alerting is service-initiated. Silence is inherently ambiguous. A monitor that is down, delayed, misconfigured, or selectively suppressing results may look exactly like a monitor that has simply found nothing new.

2.2. State-of-the-art: Chronological Transparency Logs and Monitoring

The major transparency deployments today fall into two broad families. The first family is the classic append-only log: CT[4], SumDB[5], [6], Rekor[7], and many software-transparency deployments. The second family is the authenticated directory / auditable map that underpins Key Transparency systems: CONIKS [8], SeemLESS [9], and Meta's AKD-based deployments [9] . The first family excels at public publication and append-only accountability; the second family excels at keyed lookup and, in many cases, private or semi-private membership. A verifiable index acts as a bridge: it overlays the first family with some of the second family's query power without replacing the input log as the source of truth.

Three design patterns emerge from this landscape. First, append-only logs provide strong public history integrity with concise inclusion and consistency proofs. Second, authenticated directories provide strong keyed lookup integrity, often with privacy features such as VRF-derived positions. Current practice still treats these as separate subsystems; what is mostly missing is a generic, reusable overlay that imports content-query semantics into an otherwise ordinary append-only log. That is exactly the role of a verifiable index.

The input log remains the authoritative publication structure, the verifiable index is a separate data structure, and the output log is used to make index history publicly auditable. This is conceptually different from building a full key-transparency service in front of clients and asking the log to disappear into the

background.

The design is closer to a continuous overlay over an external log prefix: it polls the latest input-log checkpoint, processes the delta, commits a new index root, and serves only the latest state plus incremental per-key deltas; historical correctness is checked by re-running the deterministic mapping logic over the input log and comparing against the output-log history.

CT is a particularly strong example because monitoring is central to the intended security outcome. RFC 9162 describes the purpose of CT as publicly logging TLS server certificates so that suspect issuance can be noticed and CA activity can be audited [10]. The log does not decide whether a certificate is legitimate for the named organization. That determination requires external ground truth: ownership or operational control of the domain, expected issuers, organizational policy, certificate inventory, and sometimes deployment context.

CT also has several features that complicate indexing. The ecosystem contains multiple independent logs. Entries may be certificates or precertificates. The same certificate can appear in several logs, and multiple entries can be created for what users may regard as the same logical certificate event. A certificate can contain multiple DNS names. Wildcard names introduce another layer of matching semantics. These details are manageable, but they mean that "index by domain" must be specified precisely rather than treated as an informal parsing step.

3. Verifiable Index Design

A Verifiable Index is an authenticated, queryable layer over an existing transparency log. Conceptually, it behaves like a back-of-the-book index: a meaningful key maps to positions in the chronological source log. The difference from a normal search database is that both the key lookup and the historical index state are cryptographically committed.

The central design insight is to separate three roles that are often conflated. The input log remains the source of truth for admission and append-only sequencing. A mapping function maps search keys to posting lists of input-log positions. An output log commits, over time, to successive index roots, thereby giving the index a public, append-only accountability trail. The Verifiable Index map hence sits between an existing input log and a newly created output log. Verifiable Indices are built on a continuous model: the overlay is updated as new log entries arrive, the service answers only against the latest committed state, and historical assurance comes from auditing the output log rather than by serving arbitrary historical snapshots to clients. [3] Importantly, the Verifiable Index does not decide whether a returned entry is legitimate. It proves what the indexed log contains under a specified mapping. A stakeholder/monitor still applies external policy or ground truth to decide whether those entries are acceptable.

3.1. Entities

A verifiable index ecosystem involves several actors, each with a distinct role in producing, organizing, verifying, and consuming transparency information.

The first actor is the claimant. This is the entity that creates a statement that should become publicly auditable. In Certificate Transparency, the claimant is typically a CA issuing a certificate or precertificate. In a binary transparency system, it may be a software publisher releasing a new package version. More

generally, a claim can be modeled as a structured object containing a type, a subject, a payload, and associated metadata. The subject identifies the entity the claim is about, while the payload contains the actual statement being made.

These claims are submitted to an append-only log. The log maintains an ordered sequence of entries and periodically publishes signed checkpoints that commit to the current log state. Each checkpoint contains the log identity, the current log size, a Merkle root over all entries up to that size, and a log signature. The log supports standard transparency-log operations, such as inclusion proofs for individual entries and consistency proofs between checkpoints. This allows clients and monitors to verify that a particular entry was included and that the log has grown append-only over time.

The indexer is responsible for transforming the raw log into a queryable structure. It reads log entries, applies a deterministic extraction function to determine which keys each entry should be indexed under, and constructs an authenticated index over those key-value mappings. For example, a certificate may be indexed under each domain name it contains, while an MTC entry may be indexed under its domain, issuer, or certificate type. The indexer then publishes an index checkpoint that binds together the index root, the corresponding source log checkpoint, and the indexing policy. Importantly, the indexer is not assumed to be trusted. Its answers must be accompanied by cryptographic proofs, and its index checkpoints must be independently verifiable.

A monitor provides this independent verification. Its role is to check that the index is faithfully derived from the underlying log. A monitor may fetch new log entries, recompute affected parts of the index, verify index update proofs, and detect whether any query-relevant entries have been omitted. In stronger deployments, monitors may recompute the full index state themselves. In lighter deployments, they may verify incremental updates and raise alerts when the indexer behaves inconsistently or incompletely. This role generalizes the function of Certificate Transparency monitors, but extends it from simply observing log entries to checking the correctness of an authenticated query layer.

A witness or cosigner can be used to make index verification practical for lightweight clients. Instead of requiring every client to verify every index update, witnesses verify the indexer's updates and then sign the resulting index checkpoints. A client can accept an index checkpoint if it carries a sufficient quorum of witness signatures according to local policy. This shifts expensive verification work to a smaller set of specialized actors while still giving clients cryptographic evidence that the index checkpoint has been checked by independent parties. Witnesses also help reduce the risk of equivocation, since an indexer that presents conflicting index histories would need to obtain conflicting witness signatures or risk being exposed.

The final actor is the querying client. This may be a browser, domain owner, package manager, wallet, relying party, or monitoring service. The client sends a query to the indexer and receives an answer together with a proof. The answer may contain matching log entries, log indices, digests, or a projection of the relevant data. The proof allows the client to verify that the answer is consistent with the authenticated index root and that the index root is tied to a particular source log checkpoint. Depending on policy, the client may also verify witness signatures, freshness constraints, source-log inclusion proofs, and anti-equivocation evidence before accepting the result.

Together, these entities separate responsibilities, avoiding full trust in any single search service. The log provides append-only publication, the indexer provides efficient query access, monitors and witnesses provide independent verification, and clients verify proofs before relying on answers. The security of the overall system depends not only on the authenticated data structures used by the indexer but also on the correspondence between the index and the source log, the freshness of accepted checkpoints, and the anti-equivocation mechanisms that ensure all clients see a consistent index history.

3.2. Verifiable Index Model

Each Verifiable Index instance is defined by a pair (L, μ). L identifies the input log. μ identifies the fixed mapping semantics: how raw leaves are parsed, canonicalized, converted to index keys, and represented as values. This distinction is important because the same chronological log can support different useful index structures depending on how its entries are interpreted and which queries the index is intended to answer.

The parameter L identifies the authoritative source of entries. A VIndex consumes the entries of this log in their original append order and maintains an authenticated index over the prefix that has been processed. The index, therefore, does not replace the transparency log or become a second source of truth. The original log remains authoritative for the existence, ordering, and contents of entries, while the VIndex provides an additional content-oriented view of those entries. A particular VIndex state should consequently be associated with a particular checkpoint of L, so that a verifier can determine exactly which log prefix the index claims to represent.

The second parameter, μ, specifies how the contents of L are transformed into indexable information. Conceptually, μ includes the complete set of rules necessary to turn a raw log leaf into one or more index updates. This includes how an entry is parsed, how relevant fields are canonicalized, which index keys are emitted, and what values are associated with those keys. For example, a mapping may associate a key with the positions of all matching leaves in the input log. In that case, the value associated with a key is naturally an ordered posting list of input-log indices. Another mapping could extract a different set of fields or construct composite keys, producing a different authenticated view over exactly the same underlying log.

It is therefore useful to regard μ as more than the implementation of a mapping function. It represents the semantic specification of the index.

This parameterization is relevant in two directions. First, a transparency ecosystem may contain multiple independent input logs. If $L1$, ..., Lm are different logs and all use the same mapping semantics μ, then they give rise to distinct VIndex instances, $VL1,\mu$, $VL2,\mu$, ..., VLm,μ.

Their posting-list positions are local to their respective input logs and must not be interpreted globally. A monitoring service that covers the whole ecosystem, therefore queries several VIndex instances and combines the verified results at a higher layer.

Second, the same input log may support multiple VIndexes if different indexing semantics are useful. Given one log L, mappings $\mu1$ and $\mu2$ produce distinct authenticated views, $VL,\mu1$ and $VL,\mu2$.

For Certificate Transparency, for example, one mapping could index entries by canonical DNS name, while another could use composite keys such as issuer and DNS name. Both operate over exactly the same CT log, but they answer different classes of queries and therefore have different completeness semantics.

This also means that μ must be treated as part of the meaning of an authenticated VIndex root. A map root by itself only commits to a key–value structure; it does not explain how the source entries were transformed into that structure. For independent reconstruction to work, different implementations must apply the same deterministic parsing, canonicalization, and key-generation rules. The mapping semantics should therefore be stable, versioned, and identifiable.

Conceptually, a published VIndex revision can be understood as binding together (L, μ, Cn, Rn), where Cn is the checkpoint of the consumed input-log prefix and Rn is the corresponding VMap root. The Output Log described later provides the durable record of these bindings. A verifier can therefore determine not only which map root is being used, but also which input log, which input-log state, and which indexing semantics that root represents.

Under this model, a VIndex is an authenticated, continuously maintained view of a particular transparency log under a particular mapping semantics. The pair (L, μ) identifies the VIndex instance, while successive roots Rn represent the evolution of that same instance as the source log grows.

3.3. Verifiable Index Architecture

The architecture translates the conceptual VIndex model from the previous section into a deployable system. The model identifies a VIndex instance by an input log L, a fixed mapping semantics μ, and a sequence of authenticated index states derived as the log grows. The architecture described here defines the components required to continuously maintain those states, publish them, and serve verifiable queries. Figure 2 shows the main components.

3.3.1 Input Log

The Input Log is the existing chronological transparency log L . A Batch Builder tracks its latest processed checkpoint, retrieves newly appended leaves, and verifies that they belong to the expected log state.

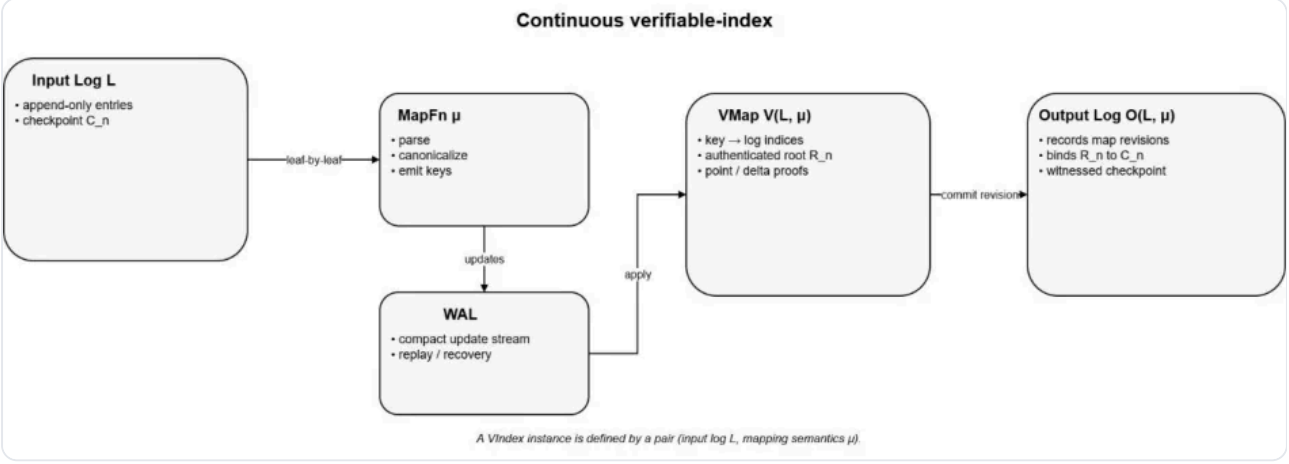


Figure 2: VIndex Architecture

The Input Log remains the source of truth. A Batch Builder observes new checkpoints and retrieves the entries that have appeared since the previously processed checkpoint. Those entries are passed through the MapFn, which implements the mapping semantics μ . The resulting key updates are persisted in a

write-ahead map transaction log (WAL) and then applied to the Verifiable Map (VMap). Periodically, the resulting map root is associated with the Input Log checkpoint from which it was derived and published as a new revision in the Output Log.

Each batch is therefore tied to a specific checkpoint Cn . The Batch Builder passes the validated entries and their stable source-log positions to the mapping stage.

For multi-log systems, each log is processed independently. Leaf position i is meaningful only together with the identity of its source log.

3.3.2 MapFn

The MapFn implements the mapping semantics μ . It deterministically transforms a source-log entry into zero or more index keys:

$(ei, i) \longrightarrow \{(k1, i), \ldots, (kr, i)\}$.

This stage includes parsing, canonicalization, key extraction, and handling of unsupported or malformed entries.

Keeping MapFn separate from the authenticated map is important: the mapping semantics determine what the index means, while the map backend determines how that state is authenticated and stored. The supplied prototype similarly treats deterministic indexing as a design requirement.

3.3.3 Write-Ahead Map Transaction Log

The output of MapFn first persisted in a write-ahead map transaction log (WAL). A WAL record associates one source-log position with the keys produced for that entry.

The WAL serves two purposes. First, it supports recovery and rebuilding without redownloading and reparsing the entire Input Log. Second, it creates a clean boundary between entry interpretation and authenticated-map maintenance.

In this project, we use this separation to ensure that different authenticated map backends consume the same derived updates.

3.3.4 Verifiable Map

The Verifiable Map stores the content-oriented view of the log. Logically, it maps an index key k to an ordered posting list of positions in the Input Log:

$k \longmapsto P(k)$.

For example, example . com $\longmapsto$ [1842, 7719, 10552, …].

The authenticated map exposes a root Rn that commits to the complete index state after processing the log up to the checkpoint Cn .

Different authenticated data structures can implement this layer. We evaluate Sparse Merkle Trees (SMT), Merkle Patricia Trees (MPT), and IAVL backends behind a common VIndex abstraction.

3.3.5 Output Log

The VMap root alone does not provide a public history of index states. The Output Log therefore records published VIndex revisions.

A revision conceptually binds (L, μ, Cn, Rn), meaning that root Rn represents input log L, under mapping semantics μ, after processing the prefix identified by checkpoint Cn .

The Output Log provides historical accountability for these map states and can itself be witnessed. The VIndex may be updated continuously while publishing revisions periodically rather than after every individual leaf.

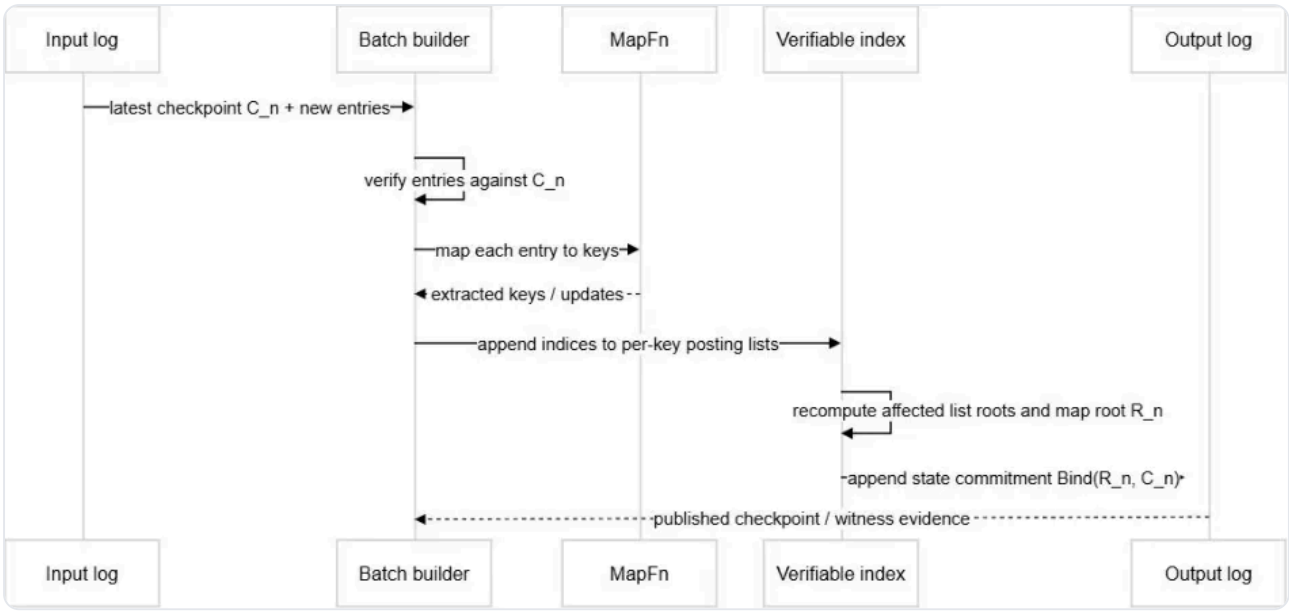


Figure 3: Verifiable Index Sequence Diagram

4. Verifiable Index for Certificate Transparency

Certificate Transparency (CT) is a particularly relevant use case for verifiable indexing. CT operates at very large scale, relies on multiple independent logs, and requires monitors to process complex X.509 objects containing multiple names, wildcard entries, precertificates, and repeated observations across logs. At the same time, the main monitoring question is naturally content-based: a domain owner wants to know which logged certificates concern names under its control.

CT can be modelled as a collection of independently indexed logs rather than as one global authenticated map. Each CT log Li has its own append-only history and therefore its own VIndex instance, VLi,μ.

This preserves the independent checkpoint sequence of each CT log and avoids introducing an artificial ordering between entries from different logs.

A CT monitoring service can then operate as an aggregation layer. For a domain query, it requests a verified result from each covered VIndex, verifies the corresponding proof chain, and merges the returned CT entries into a combined view. Completeness is therefore relative to the set of CT logs covered by the monitor and to the accepted checkpoints of those logs. The monitor can prove that it returned all matching entries from the VIndexes it queried, but this should not automatically be interpreted as completeness with respect to every possible CT log.

Mapping

The most direct CT index design would be to map DNS names to the positions of certificate or precertificate entries containing those names. For example, example . com $\longmapsto$[1250, 4821, 9317, ...].

The corresponding MapFn parses each CT leaf, extracts the relevant DNS names, canonicalizes them, and emits one key for each name. The canonicalization rules should be defined explicitly as part of the mapping semantics μ . In the CT setting, this includes how DNS names are normalized with respect to letter case and trailing dots, how internationalized domain names are represented, and how wildcard names are treated. The specification should also state which certificate fields contribute names to the index, for example, whether only the Subject Alternative Name extension is considered or whether legacy Common Name fields are included as well.

The representation of the index key also affects the type of queries the VIndex can support. A hashed exact-name key works well for direct lookups and fits naturally with sparse authenticated maps. However, it does not support suffix or subtree enumeration directly. A structured representation, such as reversed DNS labels, could make queries over subdomains easier to support, but would require more explicit ordering and range-proof semantics.

The natural value associated with a CT key is an ordered list of source-log positions. The VIndex should point back to the CT log rather than duplicate the complete certificate object inside the authenticated map. For a single CT log, a posting list can therefore be represented as $P(k)$ = [i1, i2, ..., in].

Duplicates and related CT entries

CT introduces several forms of duplication that should be distinguished rather than collapsed inside the authenticated index.

The same certificate may be present in several CT logs. A precertificate and the corresponding final certificate are related but represent different CT entries. A certificate containing several DNS names will legitimately appear under several index keys.

For this reason, the VIndex should remain close to the source-log representation and index every matching source entry. Deduplication is better performed by the monitoring layer, where the policy can be made explicit.

For example, a monitoring interface may display the same DER certificate observed in three CT logs as one logical certificate while retaining the three underlying log references. Similarly, a precertificate and final certificate can be grouped for presentation without removing either authenticated observation.

As a design principle, the VIndex should preserve the source-log evidence, while policy-dependent grouping and deduplication should be performed above the authenticated index.

Alternative VIndex views

A single exact-domain index may not support every CT monitoring use case. Because a VIndex is parameterized by both the source log and the mapping semantics, additional authenticated views can be constructed over the same CT log when needed.

Possible examples include:

VL,μ domain, VL,μ issuer+domain, VL,μ SPKI, or a structured domain index optimized for suffix queries.

These should be treated as optional views rather than as a requirement to maintain every possible index. Each additional VIndex increases construction, storage, and operational cost.

Search and monitoring

For interactive search, the client submits a domain or another supported key and receives a proof-bearing answer from each relevant VIndex. Because the request is user initiated, failures are immediately visible: the service may be unavailable, the proof may fail verification, or the answer may be bound to an unexpectedly old checkpoint.

Exact-name search is therefore the simplest use case for a CT VIndex.

Alerting requires an additional liveness mechanism. A domain owner may expect to receive a notification only when a new matching certificate appears, but silence is ambiguous: it may mean that nothing new was logged, or that the monitoring service failed.

A practical VIndex-based monitor can separate two functionalities: (i) event alerts, sent when new matching entries appear; and (ii)periodic proof-bearing heartbeats, sent even when no new matching entry exists. The client stores its most recent accepted state. An event alert contains the new matching entries together with evidence binding them to the current VIndex revision. A periodic heartbeat confirms that the monitor has advanced to a newer authenticated state even when no new entries match the subscriber's key.

If the expected heartbeat does not arrive within the agreed service interval, the client can actively query the monitoring service, query another VIndex provider, or fall back to direct inspection of the underlying CT logs.

All in all, the CT instantiation also exposes several questions that do not arise as strongly in simpler transparency systems. A deployment needs to determine how DNS names should be canonicalized, how wildcard and subdomain queries should be represented, how precertificates and final certificates should be related at the monitoring layer, and how verified results from multiple logs should be merged.

These questions are important because they affect not only performance, but also the meaning of a "complete" monitoring result. They therefore form a central part of the evaluation and open research questions.

5. Evaluation

We evaluate the VIndex prototype to quantify the trade-off between faster content-based queries and the additional cost of maintaining an authenticated index. The experiments compare a conventional append-only transparency log against three VIndex backends: a Sparse Merkle Tree (SMT), a Merkle Patricia Trie (MPT), and IAVL. All VIndex backends consume the same derived key–value updates, so the measurements primarily reflect differences in the authenticated data structures rather than in parsing or data preparation.

The experiments use real Go module data from SumDB to construct append-only logs of 2^16 through 2^24 leaves and compare a baseline log scan with SMT, MPT, and IAVL-backed authenticated indices.

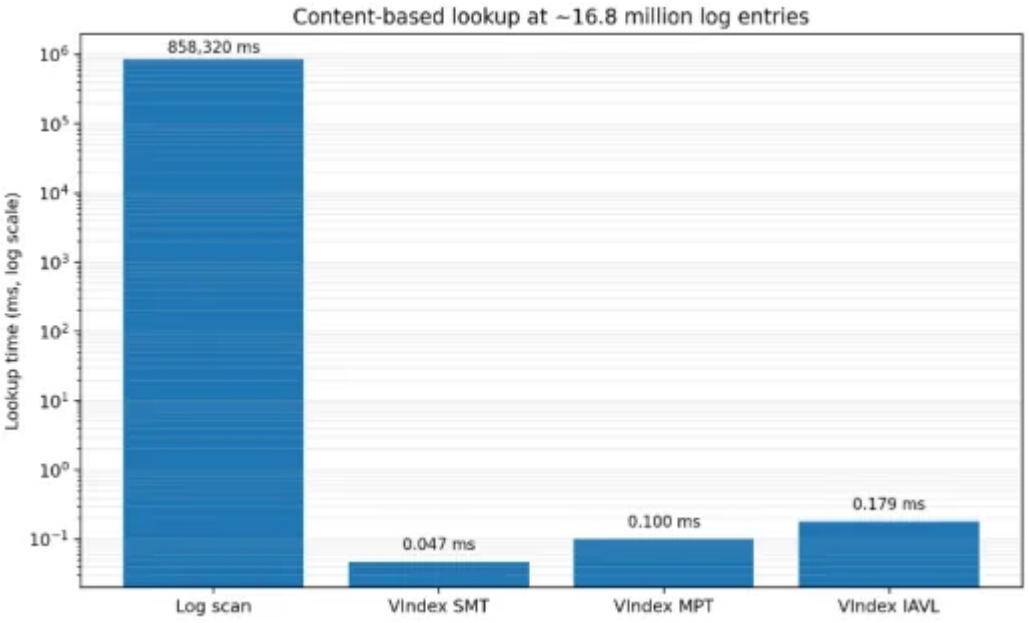
The WAL and VIndex construction costs are measured separately.

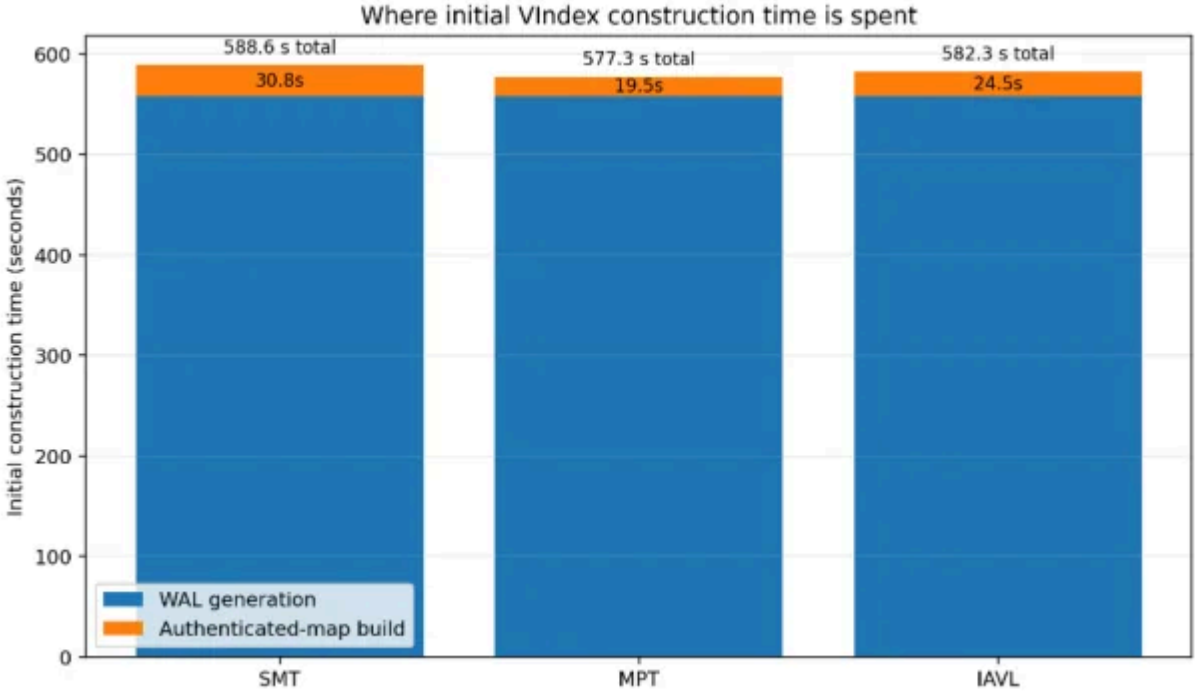
Benchmark	Reported result
Content-based lookup	Baseline scan: 858.32 s; SMT: 0.047 ms; MPT: 0.100 ms; IAVL: 0.179 ms
Inclusion-proof workflow	Baseline: 4.446 ms; best VIndex result: 3.874 ms
Authenticated-index construction	MPT: 19.53 s; IAVL: 24.51 s; SMT: 30.84 s
Full WAL generation	Approximately 558 s
Index memory footprint	SMT: 1.54 GB; MPT: 2.60 GB; IAVL: 2.94 GB
WAL storage	Approximately 2.06 GB

The benchmark results show that the main benefit of the VIndex appears for content-based aggregation queries. In the baseline design, retrieving all entries associated with a given module requires scanning the chronological log. At approximately 16 million leaves, this operation takes about 858.32 seconds. By contrast, the same lookup through the VIndex completes in 0.047 ms with SMT, 0.100 ms with MPT, and 0.179 ms with IAVL. This corresponds to a performance improvement of several orders of magnitude and demonstrates that the VIndex is particularly effective when queries would otherwise require traversing a large portion of the source log.

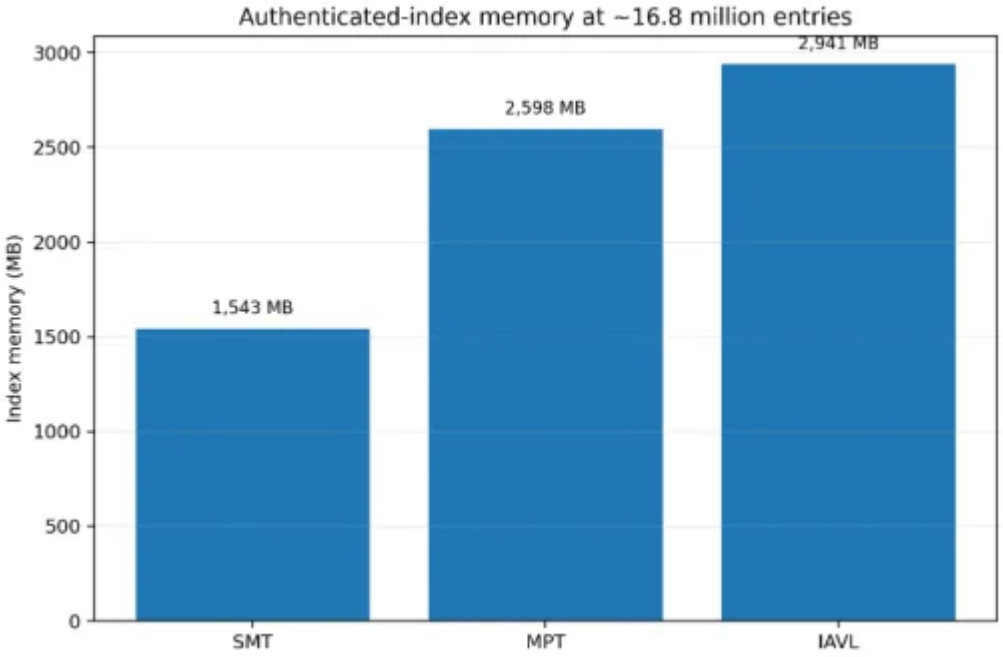
The improvement is much smaller for standard inclusion proofs. At the largest evaluated tree size, the baseline proof workflow takes approximately 4.45 ms, compared with approximately 3.87 ms for the bestperforming VIndex configuration. MPT and IAVL show similar results. This is consistent with the intended role of the VIndex: once the position of a relevant leaf is already known, the underlying Merkle proof remains efficient. The VIndex primarily reduces the cost of identifying the relevant leaf or set of leaves in the first place.

These lookup improvements come at the cost of additional index construction and maintenance. For the largest dataset, constructing the complete WAL requires roughly 558 seconds. Building the authenticated index from the WAL then takes approximately 30.84 seconds for SMT, 19.53 seconds for MPT, and 24.51 seconds for IAVL. Among the evaluated backends, MPT therefore provides the fastest construction time, while the other backends expose different trade-offs in memory use and lookup behavior.





The VIndex also introduces additional resource requirements. At approximately 16 million entries, the authenticated index consumes around 1.54 GB with SMT, 2.60 GB with MPT, and 2.94 GB with IAVL, while the corresponding WAL occupies about 2.06 GB. These measurements illustrate that the VIndex shifts part of the system cost from query time to continuous index construction and storage.



Overall, the benchmark results support the intended use of the VIndex as an additional query layer over a chronological transparency log. Its benefit is limited for operations that are already efficient in the underlying log, such as inclusion proofs for known leaf positions. The largest gains arise for repeated content-based queries that would otherwise require scanning the log. For monitoring workloads, this trade-off is particularly relevant: the system pays an ongoing cost to maintain the index, but avoids repeated full-log scans when evaluating the same or similar predicates as the log grows.

6. Conclusions:

Chronological transparency logs solve an important integrity problem: they make published events tamper-evident and auditable over time. Monitoring introduces a different problem: finding every entry relevant to a stakeholder. Today that capability is normally supplied by trusted third-party indexes. Verifiable indexing aims to retain the efficiency of those indexes while extending the transparency chain through the query layer.

The most useful architectural model is a per-instance log–map–log architecture. An Input Log remains authoritative; fixed mapping semantics μ derive keys and update operations; a continuously maintained Verifiable Index authenticates key-to-position posting lists; and an Output Log records map revisions bound to source-log checkpoints.

The evaluation results give strong evidence that the systems trade-off is worth exploring: authenticated indexes can reduce repeated aggregation scans by many orders of magnitude, although ingestion, mapping, storage, and index construction are not free. The remaining research is therefore about making the semantics reproducible, the value structures incremental, cross-layer correspondence efficient, and the monitoring experience, including alerting, end-to-end verifiable.

References

[1] “Monitors : Certificate Transparency.” Accessed: Sep. 01, 2026. [Online]. Available: https://certificate.transparency.dev/monitors/

[2] “Certificate Transparency Revisited: The Public Inspections on Third-party Monitors - NDSS Symposium.” Accessed: Sep. 01, 2026. [Online]. Available: https://www.ndsssymposium.org/ndss-paper/certificate-transparency-revisited-the-public-inspections-onthird-party-monitors/

[3] “incubator/vindex at main · transparency-dev/incubator · GitHub.” Accessed: Jun. 09, 2026. [Online]. Available: https://github.com/transparency-dev/incubator/tree/main/vindex

[4] “Certificate Transparency : Certificate Transparency.” Accessed: Nov. 11, 2024. [Online]. Available: https://certificate.transparency.dev/

[5] “Proposal: Secure the Public Go Module Ecosystem.” Accessed: Sep. 01, 2026. [Online]. Available: https://go.googlesource.com/proposal/+/master/design/25530-sumdb.md

[6] “Deploying key transparency at WhatsApp - Engineering at Meta.” Accessed: Sep. 01, 2026. [Online]. Available: https://engineering.fb.com/2023/04/13/security/whatsapp-keytransparency/

[7] “GitHub - sigstore/rekor: Software Supply Chain Transparency Log · GitHub.” Accessed: Sep. 01, 2026. [Online]. Available: https://github.com/sigstore/rekor

[8] M. S. Melara, A. Blankstein, P. University, J. Bonneau, E. W. Felten, and M. J. Freedman, “Open access to the Proceedings of the 24th USENIX Security Symposium is sponsored by USENIX CONIKS: Bringing Key Transparency to End Users CONIKS: Bringing Key Transparency to End Users,” p. 383, Accessed: Sep. 01, 2026. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technicalsessions/presentation/melara

[9] M. Chase, A. Deshpande, E. Ghosh, and H. Malvai, “SEEMless: Secure End-to-End Encrypted Messaging with less trust,” *Cryptology ePrint Archive*, 2018, doi: 10.1145/3319535.3363202.

[10] B. Laurie, E. Messeri, R. Stradling, B. Laurie, E. Messeri, and R. Stradling, “Certificate Transparency Version 2.0,” Dec. 2021, doi: 10.17487/RFC9162.