Reliable Context

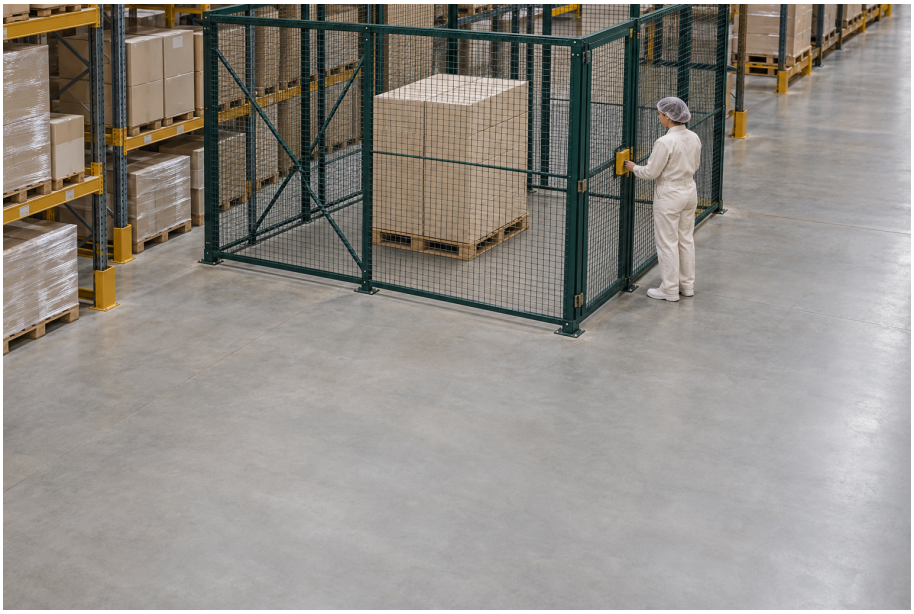
RELIABLE CONTEXT / RESEARCH & PRACTICAL GUIDANCE

Tamper-Evident Logs: Hash Chains, Merkle Trees, CT

> Reliable Context is a kernel of truth: a system of record for organizations that fuse private and public information and must defend every value they act on. One canonical record per real thing, a cited source and an observation date behind every value, unknowns declared rather than left blank, and any past state of the record reproducible.

Published by Reliable Context · 2026-10-29 · tamper evident logging · merkle tree audit log · certificate transparency merkle tree · append only log integrity · transparency log

Original article: https://reliablecontext.com/articles/tamper-evident-logs-merkle-trees-transparency-logs



In this article

1. The problem: a log is only as honest as its custodian
2. Building block: the cryptographic hash
3. The example log
4. Step 1: hash chains
5. Step 2: Merkle trees and inclusion proofs
6. Step 3: consistency proofs and the append-only property
7. Comparing the three designs
8. Split views, gossip and witnesses
9. Real systems
10. Trusted timestamps and anchoring
11. What integrity proofs do not prove
12. Design choices for a records platform

The problem: a log is only as honest as its custodian

An audit log is meant to answer, after the fact, what happened and in what order. Ordinary logs are files or database tables. Whoever controls the storage can edit a line, delete a line, or insert one with a back-dated timestamp, and a later reader has no way to tell. Access controls reduce who can do that; they do not make it detectable when someone does.

A **tamper-evident log** is a log structure in which any modification of past entries can be detected by someone holding a small piece of earlier information about the log. It is not a **tamper-proof** log. Crosby and Wallach, whose 2009 paper is the standard reference, put it plainly: "It is impractical to prevent the destruction or alteration of digital records that are in the custody of a Byzantine logger", and "tamper-evidence requires auditing. If the log is never examined, then tampering cannot be detected" (Crosby and Wallach, 2009).

This article builds the idea up in three steps, with computed values at each step: a hash chain, a Merkle tree with inclusion proofs, and a Merkle log with consistency proofs, which is what Certificate Transparency and Sigstore's Rekor use. It ends with what these proofs cannot establish, which matters as much as what they can.

Building block: the cryptographic hash

A **cryptographic hash function** maps any input to a fixed-length digest such that it is computationally infeasible to find two inputs with the same digest or to alter an input without changing its digest. NIST's Secure Hash Standard, FIPS 180-4, specifies the SHA-2 family, including SHA-256, and states that the digests "are used to detect whether messages have been changed since the digests were generated" (FIPS 180-4). Every value in this article is SHA-256, shown as the first 12 hexadecimal characters.

A digest is also a name. Storing an object under the hash of its content is called content addressing: if the content changes, so does its name.

The example log

The worked example uses four entries from a fictional quality system at Halden Therapeutics AG (fictional company, fictional people):

Index	Entry (UTF-8 bytes, exactly as hashed)
0	`2026-03-02T09:14Z set batch HT-0412 status=released by qa.mueller`
1	`2026-03-02T11:40Z set batch HT-0413 status=quarantined by qa.ito`
2	`2026-03-03T08:05Z set supplier S-77 lei=5299000EXAMPLE00001 by md.svensson`
3	`2026-03-03T16:22Z set batch HT-0413 status=released by qa.ito`

The tampering scenario: someone wants entry 1 to read `status=released` instead of `status=quarantined`, so that batch HT-0413 appears never to have been held.

Step 1: hash chains

A **hash chain** links each entry to everything before it. Define `c[-1]` as 32 zero bytes and, for each entry `e[i]`:

```
c[i] = SHA-256(c[i-1] || e[i])
```

where `||` is byte concatenation. The last value, `c[3]`, is a commitment to the whole log: change any byte of any entry and `c[3]` changes.

i	c[i] (original)	c[i] after editing entry 1
0	`dbd5476d76f0`	`dbd5476d76f0`
1	`c075aeed1a22`	`9198a9e1158e`
2	`6e7ca10dc313`	`afe7cb5feeae`
3	`5bea68c7757b`	`b2bc091c48f1`

If an auditor recorded `c[3] = 5bea68c7757b` on 3 March, the edit is detectable: recomputing the chain from the stored entries gives `b2bc091c48f1`.

Three limits show why hash chains are only a starting point.

1. **Someone must hold an earlier value outside the logger's control.** If the logger can also replace the recorded `c[3]`, it can recompute the whole chain after the edit. The chain makes tampering evident only to someone who kept, or can obtain from an independent party, a value published before the edit.
2. **Proving one entry costs the whole prefix.** To show that entry 1 is in a log whose head is `c[3]`, the verifier must recompute every link from entry 1 forward. Crosby and Wallach note that "a classic hash chain might require an 800 MB trace to prove that a randomly chosen event is in a log with 80 million events", where their tree-based structure "returns a 3 KB proof with the same semantics" (Crosby and Wallach, 2009).

3. **Truncation of the tail is invisible to anyone who has not seen the later head.** Deleting entries 2 and 3 leaves a valid chain ending at `c[1]`.

Earlier secure-logging schemes worked on confidentiality and key management around such chains; Crosby and Wallach cite Schneier and Kelsey's 1999 scheme, in which the logger encrypts entries with keys derived by iterated hashing and destroyed after use (Crosby and Wallach, 2009). The proof-size limits above are what their tree-based design addresses.

Step 2: Merkle trees and inclusion proofs

A **Merkle tree** is a binary tree in which each leaf is the hash of a data entry and each internal node is the hash of its two children. The root hash commits to every entry. RFC 9162, the specification of Certificate Transparency version 2.0, defines the tree used by CT logs precisely (RFC 9162, section 2.1.1):

- Leaf hash: `HASH(0x00 || d[i])`
- Internal node: `HASH(0x01 || left || right)`
- For n > 1 entries, split at k, the largest power of two smaller than n, and hash the two subtrees.

The different prefixes, `0x00` for leaves and `0x01` for nodes, are **domain separation**: the RFC states that this "is required to give second preimage resistance", so that an internal node can never be passed off as a leaf (RFC 9162). The RFC also notes that its tree "is essentially the same as the history tree proposed by [CrosbyWallach], except our definition handles non-full trees differently."

For the four example entries:

```
                root  300654b706f7
               /                  \
      n01  b3092df1ab6a        n23  5876c4d0855f
       /          \             /          \
L0 50a335ebafe5  L1 21673c4cfb82  L2 ffb2017ba41c  L3 24afd0016804
```

An **inclusion proof** (Crosby and Wallach call it a membership proof) is, in RFC 9162's words, "the shortest list of additional nodes in the Merkle Tree required to compute the Merkle Tree Hash for that tree" from a given leaf (RFC 9162, section 2.1.3). To prove entry 2 is in the tree with root `300654b706f7`, the log returns two hashes:

Step	Verifier computes	Using proof element
1	`L2 = HASH(0x00	
2	`n23 = HASH(0x01	
3	`root = HASH(0x01	

The computed root matches the published root, so entry 2 is in the log. The proof has one hash per level of the tree, so its size grows with the logarithm of the log size: about 20 hashes for a million entries, 27 for a hundred million. That is the efficiency gain over the hash chain.

Now apply the tampering scenario. Editing entry 1 changes `L1`, therefore `n01`, therefore the root, which becomes `c31d92c1fa81`. Anyone holding the earlier root `300654b706f7` and asking for any inclusion proof that passes through `n01` will find a mismatch.

Step 3: consistency proofs and the append-only property

Inclusion proofs show that an entry is in the tree a log presents today. They do not show that today's tree extends yesterday's. A dishonest log could rebuild the tree from scratch with entry 1 edited and publish a new root; each inclusion proof against the new root would verify.

A **consistency proof** closes that gap. RFC 9162 states that "Merkle consistency proofs prove the append-only property of the tree": given an earlier root over the first m entries and a later root over n entries, the proof is the list of nodes needed to verify that the first m entries are identical in both trees (RFC 9162, section 2.1.4). Crosby and Wallach call the same idea an incremental proof, one that shows "current commitment and prior commitment make consistent claims about past events" (Crosby and Wallach, 2009).

In the example, suppose an auditor saw the log on 3 March at 09:00, when it held three entries, and recorded the root `8184b469f860`. Later the log has four entries and root `300654b706f7`. Running the RFC 9162 proof-generation algorithm for m = 3, n = 4 yields three hashes: `L2 = ffb2017ba41c`, `L3 = 24afd0016804`, `n01 = b3092df1ab6a`. The auditor checks two things with them:

Check	Computation	Result
Old root is reproduced	`HASH(0x01	
New root is reproduced from the same nodes	`HASH(0x01	

Because both roots are rebuilt from the same `n01` and `L2`, the first three entries are unchanged. If entry 1 had been edited, `n01` would differ and the old root could not be reproduced.

Together, the two proof types give the semantics Crosby and Wallach set out: the logger "must be able to prove that individual logged events are still present, and that the log, as seen now, is consistent with how it was seen in the past" (Crosby and Wallach, 2009).

Comparing the three designs

Property	Plain log	Hash chain	Merkle log with proofs
Edit of a past entry detectable	No	Yes, by a holder of a later head	Yes, by a holder of any later root
Proof that one entry is present	Not available	Linear in log length	Logarithmic (inclusion proof)
Proof that the log only grew	Not available	Linear recomputation	Logarithmic (consistency proof)
Tail truncation detectable	No	Only by a holder of a later head	Only by a holder of a later root
Presenting different logs to different readers	Undetectable	Undetectable without comparison	Undetectable without comparison

The last row is the subject of the next section.

Split views, gossip and witnesses

Every proof above is checked against a root the verifier already trusts. A log that shows one history to one auditor and another history to a second auditor can produce valid proofs for each. This is called a **split view**, or equivocation. RFC 9162 lists "presenting different, conflicting views of the Merkle Tree at different times and/or to different parties" among the ways a log can misbehave, and says such violations "can be detected by multiple clients comparing their instances of the STHs", a technique "known as 'gossip'", which it describes as "an active area of research and not defined here" (RFC 9162, section 11.3). STH stands for signed tree head: the log's signature over its tree size and root hash.

Crosby and Wallach make the same structural assumption from the auditing side: "At least one auditor is assumed to be incorruptible" (Crosby and Wallach, 2009). The practical lesson is that a transparency log's guarantees come from independent parties who record and compare its roots. Without them, the log is a well-structured file controlled by one party.

Real systems

Certificate Transparency (CT). CT logs record TLS server certificates so that anyone can "audit certification authority (CA) activity and notice the issuance of suspect certificates as well as to audit the certificate logs themselves" (RFC 9162, abstract). The original protocol is RFC 6962 (2013); RFC 9162 (December 2021, Experimental) obsoletes it (RFC 6962; RFC 9162). A log commits to a Maximum Merge Delay, the time within which a submitted certificate must appear in the tree, and clients check that commitment by requesting inclusion proofs.

Sigstore Rekor. Rekor is the transparency log in Sigstore's software-signing infrastructure. Its documentation describes it as providing "an immutable, tamper-resistant ledger of metadata", built on a verifiable data structure that lets auditors check that "the log remains append-only and entries are never mutated or removed", and points to independent witnessing and monitoring tools for doing so (Sigstore, Rekor overview).

Both systems separate three roles: the log operator, who appends and serves proofs; monitors or auditors, who check consistency over time; and clients, who check inclusion of the entries they care about.

Trusted timestamps and anchoring

A Merkle log proves order and integrity, not wall-clock time. The timestamps inside the example entries are whatever the writer typed. Two techniques add external evidence of time.

RFC 3161 time-stamping. A Time Stamping Authority (TSA) signs a token binding a hash to a time. RFC 3161 states that a time-stamping service "supports assertions of proof that a datum existed before a particular time", and that the request's `messageImprint` field should contain the hash of the datum (RFC 3161). Time-stamping a log root at regular intervals bounds when the entries beneath it existed. It says nothing about when they were created, only that they existed no later than the token's time.

Anchoring. Publishing a root periodically somewhere the logger does not control, such as a separate organization, a regulator's portal or another transparency log, creates an independent copy of the commitment. This is the same idea as gossip, applied to a single organization's logs.

What integrity proofs do not prove

A verified inclusion proof establishes one thing: this exact byte string is at this position in a log whose root is this value. It does not establish any of the following.

Question	Answered by the Merkle proof?	What would answer it
Was the entry's content true?	No	Evidence for the claim itself: source, method, review
Was the writer authorized to make it?	No	Access control records, signatures checked against an authorization policy
Did the event happen at the time written in the entry?	No	Trusted timestamps, independent corroboration
Was anything left out of the log?	No	Completeness controls at the point of capture
Is the log the same one other readers see?	Only with gossip or witnesses	Independent comparison of roots

Crosby and Wallach address the second row directly: "a malicious client could insert garbage, but we wish to ensure that an event, once correctly inserted, cannot be undetectably hidden or modified" (Crosby and Wallach, 2009). Integrity of the record is distinct from correctness of what it records. In the example, if `qa.ito` mistakenly typed `quarantined` for a batch that was fine, the Merkle log will faithfully preserve the mistake. The correction should be a new entry, not an edit, and the history will then show both.

The same distinction appears in signed data exports: a signature shows who endorsed a bounded data set and that it has not changed, not that it is right. The live article on signed RDF exports works through that case for graph data.

Design choices for a records platform

For teams that keep regulated records or an append-only log of changes to them, the material above reduces to a short set of decisions.

1. **Canonical entry encoding.** The hash covers bytes, not meaning. Fix one serialization for entries (field order, character encoding, timestamp format) before hashing, or two equivalent entries will hash differently.
2. **Tree, not only chain.** Use a Merkle structure if any party will need to prove that a single entry is present without handing over the whole log.
3. **Publish roots on a schedule, outside your control.** A root that only the logger holds protects against nothing the logger does.

4. **Serve consistency proofs on request.** Anyone holding an old root should be able to verify that the current log extends it.
5. **Corrections are appends.** Never edit an entry; record a new entry that supersedes it and references the old one. The chain of custody is the full sequence, including errors.
6. **Time-stamp roots, not entries.** One RFC 3161 token over a root covers every entry beneath it.
7. **State the limits.** Document that the log proves integrity and order, not truth or authorization, and say which other controls cover those.

Integrity proofs answer "has this changed since it was recorded?"; whether it was true when recorded is a separate question that the record itself must carry evidence for, which is the problem Reliable Context addresses.

Source links

Links cited in this article, in order of first appearance. Full addresses are provided for readers using extracted text.

1. https://www.usenix.org/legacy/event/sec09/tech/full_papers/crosby.pdf
2. https://csrc.nist.gov/pubs/fips/180-4/upd1/final
3. https://www.rfc-editor.org/rfc/rfc9162
4. https://www.rfc-editor.org/rfc/rfc6962
5. https://docs.sigstore.dev/logging/overview
6. https://www.rfc-editor.org/rfc/rfc3161

THE PUBLISHER

About Reliable Context

Reliable Context is a system of record for organizations that fuse private and public information and must defend every value they act on.

The guarantees

One canonical record per real thing: duplicates resolve by rule, ambiguity is shown rather than hidden, and related organizations relate without ever being merged. A verbatim, cited source and an observation date sit behind every value. Unknowns are declared as not found, blocked or not applicable with a reason, never left as an empty cell. Any past state of the record is reproducible, together with what changed since.

Where it runs

It runs inside the customer's perimeter, one database per organization. Permitted sources, model routing, reader clearance and retention are set as the customer's own policies, and every value is exportable with its chain of custody for an auditor or regulator.

What it is not

It is not a vector database or a RAG framework, not a data catalog, MDM or data-quality suite, not a CRM or data-enrichment vendor, and not a chatbot or AI assistant. Where a reader needs one of those, the honest answer is to say so and point them there.

Visit

See Reliable Context.

Reliable Context is in production use in two sectors; more than six thousand of its identity decisions were made by rule and none has been reversed. Articles published here explain general practice, cite their primary and academic sources, and name the jurisdiction of any regulatory rule they describe. They are not legal, regulatory or compliance advice for a specific organization, and requirements change.

Website: https://reliablecontext.com

This article is educational. Verify consequential medical, legal, financial, regulatory and technical decisions with appropriate professionals and the cited primary sources. Company services are described separately from the article's research findings.