



SERIES

THE RECORD · SIX PAPERS · OPEN LICENCE

The Project *Handshake.*

Working notes on tamper-evident record-keeping

PAPERS

PAPER 001	Assume the store is already lost	THREAT MODEL · 9 MIN
PAPER 002	Two trees	CONSTRUCTION · 14 MIN
PAPER 003	A signature that knows where it lives	CONSTRUCTION · 12 MIN
PAPER 004	What verification cannot see	LIMITS · 11 MIN
PAPER 005	Three questions, one substrate	ARCHITECTURE · 13 MIN
PAPER 006	What a revocation cannot recall	WITHDRAWAL · 15 MIN

ISSUED

six papers · published 31 August – 1 September 2026 · CC BY 4.0
digital1.foundation/articles/the-project-handshake/

Stichting Digital One Foundation i.o. · Amsterdam

These notes describe work contributed to an entity in formation.

Contents

001

Assume the store is *already lost*

THREAT MODEL · 9 MIN · PUBLISHED 31 AUGUST 2026

Prevention and detection are different security properties. What a record can prove once the operator is inside the adversary.

§ 1The question a record actually has to answer5

§ 2Naming the adversary, including the awkward one6

§ 3Why access control is necessary and not sufficient6

§ 4Three requirements detection imposes7

§ 5What this cannot do, stated before anyone else says it8

§ 6Why this is becoming a practical question, not a theoretical one9

§ 7What follows from all this10

References10

002

Two *trees*

CONSTRUCTION · 14 MIN · PUBLISHED 31 AUGUST 2026

A record you can still check after you remove part of it — and the length-encoding defect that let a value be changed without changing its commitment.

§ 1The problem with hashing a document11

§ 2Two false starts worth walking through12

§ 3The first tree: over the fields of one record13

§ 4The defect: when two different records have the same commitment13

§ 5Withholding, and why the label needs its own commitment15

§ 6The second tree: over records, doing a different job16

§ 7Two details of RFC 6962 that implementations get wrong16

§ 8What is deliberately not decided here17

References18

003

A signature that knows *where it lives* 19

CONSTRUCTION • 12 MIN • PUBLISHED 31 AUGUST 2026

Deriving the signed bytes from the replay attacks they have to survive, one field at a time.

§1	Derivation, not decoration	19
§2	Step 0 — sign the record hash	19
§3	Step 1 — bind each record to the one before it	20
§4	Step 2 — bind the position	20
§5	Step 3 — bind the domain	20
§6	Step 4 — bind the owner (the one that was missed)	21
§7	Migrating a signature format without re-signing anything	22
§8	Why Ed25519	23
§9	Two failures that verifiers routinely treat as non-failures	23
§10	The result	25
	References	25

004

What verification *cannot* see 26

LIMITS • 11 MIN • PUBLISHED 31 AUGUST 2026

A truncated chain is a valid chain. On the limits of self-checking evidence, and why the honest answer to “why not a blockchain” is a witness.

§1	The prefix problem	26
§2	The commitment, and why it must be held elsewhere	27
§3	A commitment cannot validate a slice	27
§4	“Why not a blockchain?”	29
§5	The gap that remains: split view	29
§6	Time is a separate problem, and it needs the same kind of witness	30
§7	An honest ledger of what is open	31
	References	32

005

Three questions, *one substrate*33

ARCHITECTURE · 13 MIN · PUBLISHED 1 SEPTEMBER 2026

Governance, liability evidence and agent authority are three layers, not one. What they share, how they compose, and the seams still open.

§1	Three questions that look like one	33
§2	Why they have to be separate layers	34
§3	What they share, and why sharing is safe	35
§4	How they compose over one event	36
§5	The seams, including the one that is not finished	37
§6	What the layering costs	38
§7	Provenance, and what is out of scope	39
	References	39

006

What a revocation *cannot recall*40

WITHDRAWAL · 15 MIN · PUBLISHED 1 SEPTEMBER 2026

Revocation is a claim about which actions can still commit, not a state change. Where that set is decided, and the floor custody puts under it.

§1	The question people ask, and the question they mean	40
§2	Blast radius is a specification obligation	41
§3	Absence is not health	42
§4	The interval, honestly	43
§5	The commit boundary	44
§6	The widest lever needs two hands	45
§7	What withdrawal cannot recall	46
§8	Provenance, and what is out of scope	47
	References	48

END

Colophon49

SOURCES AND THEIR HASHES, LICENCE, AND HOW THIS DOCUMENT IS SET

Assume the store is *already lost*.

Prevention and detection are different security properties, they answer to different adversaries, and treating them as interchangeable is the most common error in evidential record-keeping. This paper states the threat model the rest of the series is designed against — including the part where the party writing the records is inside it.

PAPER • 001 PUBLISHED • 31 AUG 2026 READING • 9 MIN

LICENCE • CC BY 4.0

Canonical • digital1.foundation/articles/the-project-handshake/001-assume-the-store-is-lost.html

§1

The question a record actually has to answer

An operational record — of a decision a system made, of the state it was in when it made it — is usually built to answer the question *what happened?* That is the wrong question, or rather it is the easy half of the right one. The question that matters arrives later, from someone with an interest in the answer being different, and it is:

THE OPERATIVE QUESTION

Can a party who has no reason to trust you satisfy themselves that this record is the one that was written at the time — *without needing your cooperation to do it?*

Every clause there is load-bearing. A party who has no reason to trust you, because the moment a record is contested, the credibility of whoever produced it is precisely what is in dispute. Satisfy themselves, because an assurance you provide is a claim, not evidence. Without your cooperation, because a check that requires your software, your servers or your permission is not a check; it is a request to be trusted, wearing the costume of a verification.

Systems designed to answer *what happened* and systems designed to answer that question look superficially similar — both write rows, both keep timestamps — and they differ almost entirely in their threat model.

§ 2

Naming the adversary, including the awkward one

Threat models are usually stated too politely to be useful. Here is a blunt one. The adversary is any party who can write to the store where the records live. In practice that set includes:

- an attacker who has obtained database credentials, which is the case everyone models;
- an administrator acting outside policy, which is the case everyone has and few model;
- a migration script written under time pressure, which is the case nobody calls an adversary and which causes more silent corruption than the first two combined;
- a party under legal or commercial pressure for the record to say something else;
- and — the one that is usually omitted — **the operator of the system itself.**

That last inclusion is the whole design. If the operator is outside the threat model, then every guarantee in the system reduces to "the operator says so", and the record is worth exactly the operator's reputation at the moment it is questioned. Which is to say: worth least precisely when it is needed most.

Placing yourself inside your own threat model is uncomfortable, and it forces a specific discipline. It means no verification step may depend on a secret only you hold, no check may be performed only by your code, and any property you cannot demonstrate to a hostile reader is a property you should stop claiming. Most of the constructions in this series are what remains after applying that filter.

§ 3

Why access control is necessary and not sufficient

The instinctive answer to this problem is to make the store unwritable: restrict permissions, put the data somewhere append-only, revoke `UPDATE` and `DELETE`. Do all of that — it is genuinely valuable and it raises the cost of the easy attacks. But understand what class of property it is.

Access control is a **preventive** control, and preventive controls share a failure mode: when they fail, they fail **silently and indistinguishably from normal operation**. A permission that was revoked and is now, for some reason, not revoked, produces no signal. A row rewritten by someone who legitimately holds the credential looks exactly like a row written legitimately. The system continues reporting that everything is fine, because from inside the system everything **is** fine.

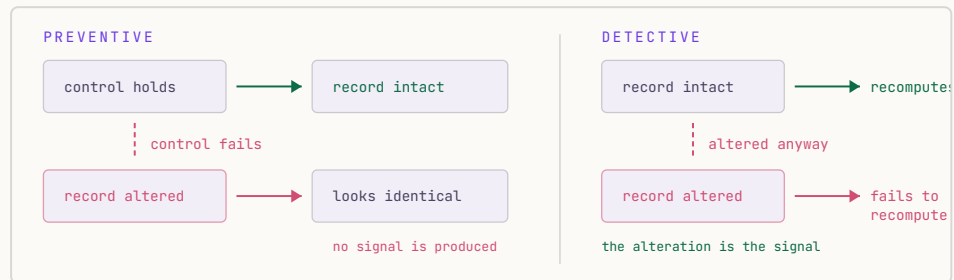


FIGURE 1 The asymmetry that motivates everything else. A preventive control that fails produces a state indistinguishable from success. A detective construction produces a signal precisely in the case the preventive control was supposed to stop.

Detection is what still works **after** the preventive control has failed. It does not compete with access control; it is the layer that assumes access control lost. The two together give you a system where the cheap attacks are blocked and the expensive ones are visible — which is the best available outcome, because no arrangement of software makes a database physically unwritable by someone holding its credentials.

A PRECISION WORTH INSISTING ON

The correct term for the resulting property is **tamper-evident**, and the distinction from the more marketable adjective is not pedantry. One says **this cannot be altered**, which is false of any row in any database. The other says **an alteration cannot be made quietly**, which is achievable and is what an examining party actually needs. A vendor who uses the stronger word has, in a single adjective, told you they have not thought about the failure mode.

§ 4

Three requirements detection imposes

Once the goal is detection under the adversary in §2, three requirements follow, and each of them eliminates a large class of otherwise reasonable designs.

1 • The record must prove itself, not match a copy

A tempting design keeps a second copy of each record — a checksum table, a shadow log, a replica — and verifies by comparison. This does nothing against the adversary in §2, because a party who can write the

record can write the copy. Comparison detects accidental divergence, which is a real and different problem, and it detects deliberate alteration only if the attacker forgets one of the two places.

What is needed instead is **re-derivation**: the check must recompute the record's identity from the record itself, using a function anyone can run, and compare against a value that is bound to that computation. Nothing is trusted because it was stored; things are trusted because they recompute.

2 · The check must be runnable by someone who does not have us

This requirement is the one that most changes engineering decisions, because it rules out anything convenient. It means the verification algorithm must be describable in a page, implemented in primitives with independent implementations, and executable with no network access, no accounts and no proprietary software. In practice it means standard cryptographic primitives, an unambiguous byte layout, and a verifier small enough that a sceptic can read it in an afternoon and rewrite it in another language to confirm it agrees.

It also means resisting a very natural feature: a verification endpoint. An API that reports whether a record is valid is worse than useless in a dispute, because its answer is the operator's assertion once again, now with an HTTP status code attached.

3 · A failure must localise

"Something in this set of records is wrong" is nearly worthless to an examining party. The check must say **which** record failed and **which property** failed of it — content, position, or attribution — because those three failures have different explanations and different consequences. Designs that hash a whole collection into one value satisfy the first two requirements and fail this one, which is a large part of why the constructions in Paper 002 are trees rather than digests.

§ 5

What this cannot do, stated before anyone else says it

A paper that only lists what a construction achieves is marketing with citations. The limits are more informative than the capabilities, and they are structural rather than temporary:

IT DOES NOT	WHY NOT, AND WHAT DOES
Prevent alteration	Nothing at this layer can. Preventing writes is the job of access control, database-level constraints and operational separation of duties. This layer assumes those have failed.
Recover the original	Detection tells you a record no longer matches what was recorded. It does not restore it. That is backup and retention, which are separate disciplines with separate failure modes.
Establish that a record is true	A perfectly verified record of a false statement is a perfectly verified false statement. These constructions bind a record to a moment and an author; they say nothing whatsoever about whether the record describes reality.
Establish <i>when</i> , by itself	A signature proves authorship, not time. Establishing time requires a third party who has no interest in the answer. That is the subject of Paper 004, and it is the least complete part of most systems, including honest ones.
Tell you what an examining body will conclude	Whether a record is given weight in any particular forum is a question for that forum. A construction can make a record checkable; it cannot make it persuasive, and any claim to the contrary is a claim about someone else's decision.

§ 6

Why this is becoming a practical question, not a theoretical one

The engineering above is not new; transparency logs and hash-linked records have existed for years. What is changing is the frequency with which an organisation is asked to produce an operational record to a party that has no reason to be generous about it.

One concrete example, offered for context rather than as legal analysis. Directive (EU) 2024/2853 on liability for defective products – which by Article 4(1) treats software as a product, and whose recital 13 addresses AI systems directly – applies from **9 December 2026**. Its Article 9(1) requires Member States to ensure that a defendant can be ordered to disclose relevant evidence at its disposal, subject to conditions of necessity and proportionality in Article 9(3) and to the protection of confidential information and trade secrets in Article 9(4). Article 10(2)(a) provides that where a defendant *fails* to disclose under Article 9(1), defectiveness is presumed – a presumption the defendant may rebut under Article 10(5). Recital 46 states the purpose plainly: the presumption exists to give an incentive to comply with the disclosure obligation. ^[1]

Read that as an engineer rather than a lawyer and it describes a system requirement. A record must be *producible* on demand, *partial* – because withholding what is irrelevant or confidential is legitimate and contemplated by the text – and *checkable by the other side*, because a

disclosure the recipient must simply believe is unlikely to end the argument. Those three requirements together are not satisfied by an ordinary application log, and the tension between the second and the third is the subject of the next paper.

A NOTE ON SCOPE

Legal instruments are cited here to explain why a technical property is worth building, and to their published text so a reader can check the wording. Which obligations attach to a given organisation, in a given jurisdiction, for a given system, is a question for its advisers. These are engineering notes.

§7

What follows from all this

Everything in the remaining papers is forced by the three requirements in §4 rather than chosen for elegance, and it is worth previewing why, because the constructions look arbitrary until you see the attack each one answers.

A record must prove itself, so its identity is derived from its content by a function anyone can run. A disclosure must remain checkable after parts are removed, so that identity cannot be a single digest over the whole — it has to be a tree over independently committed fields, and the leaves have to be salted or a low-entropy withheld value is recoverable by guessing. Failures must localise and attribution must survive, so a signature has to cover not just content but position and ownership. And because none of that detects the removal of records that were never handed over at all, something outside the system has to hold a commitment to what the whole was.

Each of those sentences is a paper. They were also, in order, four opportunities to get something subtly wrong — and the most useful parts of what follows are the places where an earlier version of this work did.

References

- 1 Directive (EU) 2024/2853 of the European Parliament and of the Council of 23 October 2024 on liability for defective products, OJ L, 2024/2853, 18.11.2024. Articles 4 (1), 9, 10 and recitals 13 and 46. Cited for context; not legal analysis.
- 2 NIST, **Secure Hash Standard (SHS)**, FIPS PUB 180-4, August 2015. The SHA-256 function referenced throughout the series.
- 3 B. Laurie, A. Langley, E. Kasper, **Certificate Transparency**, RFC 6962, June 2013. The Merkle tree construction and its two proof types, discussed in Papers 002 and 004.

Two *trees*.

A document hash forces a choice nobody should have to make: disclose everything, or disclose something unverifiable. The way out is a Merkle tree over **fields** rather than a digest over bytes — plus a second tree over **records**, doing an entirely different job. This paper gives both constructions at byte level, and the encoding defect that made an earlier version of the first one accept a value that had been changed.

PAPER · 002 PUBLISHED · 31 AUG 2026 READING · 14 MIN

LICENCE · CC BY 4.0

Canonical · digital1.foundation/articles/the-project-handshake/002-two-trees.html

§1

The problem with hashing a document

Take the ordinary approach. A record is serialised, hashed, and the digest is signed. Anyone can recompute the digest and check the signature. This is correct, standard, and it satisfies the first two requirements from [Paper 001](#).

Now remove one field, because it names a third party, or reveals a commercial term, or is simply irrelevant to the matter at hand. The digest no longer recomputes. The recipient's only options are to take the redaction on trust — which returns us to the operator's word — or to demand the unredacted record, which forces disclosure of things that were legitimately withheld.

This is not an exotic corner case. It is the normal shape of disclosure: relevance is scoped, confidentiality is protected, and both of those are contemplated by the instruments that compel production in the first place.^[1] A construction that treats any withholding as tampering has misunderstood what it is for.

THE REQUIREMENT, PRECISELY

A record's identity must be ***stable under removal***: fixed when the record is written, and still reproducible from a disclosure in which some fields are absent — without the recipient learning the absent values, and without them having to accept the sender's account of what was removed.

§2

Two false starts worth walking through

Hash each field independently

The obvious repair: commit to $H(\text{value})$ per field. Withholding then means disclosing the hash instead of the value, and everything else still checks.

This fails immediately, and it fails worst on exactly the fields most worth withholding. A hash conceals a value only insofar as the value is unguessable. Consider a withheld boolean: an examining party computes $H(\text{true})$ and $H(\text{false})$ and reads the answer. A withheld severity from a five-value enumeration takes five hashes. A withheld monetary amount known to be a round figure in a plausible range takes a few million — that is, under a second. The withheld field is only as private as its value space is large, and low-entropy fields are the common case.

Salt each field, and stop there

So salt: commit to $H(\text{salt} \parallel \text{value})$ with a fresh random salt per field, disclosed alongside the value and withheld alongside it. Now the guessing attack fails, because the adversary cannot compute a candidate hash without the salt.

But nothing binds the fields into a record. Given a bag of independent commitments, a discloser can drop one entirely and the remainder still verifies — because there is no statement anywhere about how many fields there were or which. Deletion becomes invisible, which is precisely the property we were trying to obtain.

What is needed is a structure that (a) commits to the set as a whole, (b) lets an individual leaf be replaced by its hash without disturbing the whole, and (c) makes an absence visible as an absence. That structure is a Merkle tree, and the leaves are fields.

§3

The first tree: over the fields of one record

Order the record's top-level fields by path. For each, generate a random salt and compute a leaf. The record's identity is the Merkle root over those leaves, computed per RFC 6962. ^[2]

The leaf is where the care goes:

```
// one field's commitment – every component length-prefixed
leaf(path, value, salt) =
  SHA-256( 0x02
    || u32be(len P) || P      P = utf8(canonical(path))
    || u32be(len S) || S      S = 16 raw salt bytes, exactly
    || u32be(len V) || V )    V = utf8(canonical(value))
```

The `0x02` prefix separates a field leaf from the tree's own `0x00` leaf and `0x01` interior-node prefixes, so no value in one domain can be presented as a value in another. **canonical** is deterministic JSON with keys sorted recursively – so the same facts always produce the same bytes, and formatting can never break a check.

Note that the path is **canonicalised** too, which means it is JSON-encoded, quotes included. That is not decoration: it makes the path a self-delimiting encoded value rather than a raw string, which matters for the same reason the length prefixes do.

§4

The defect: when two different records have the same commitment

The construction above is the corrected one. The first version was this, and the difference is easy to miss:

```
// the earlier, defective form
leaf(path, value, salt) = SHA-256( 0x02 || P || S || V )
// with the salt validated as "at least 16 bytes"
```

No length prefixes, and a minimum salt length rather than a fixed one. Both of those look harmless. Together they are a collision, and it is available to the party who writes the record – which, per Paper 001, is the adversary we care about.

DEFECT · FOUND BY ADVERSARIAL REVIEW, 31 AUGUST 2026

A value can be shortened by lengthening its salt

Because the components are concatenated with no separator and no length, the boundary between the salt and the value is not recoverable from the hashed bytes. An author who chooses the

salt can move the boundary.

Take a field whose value is the number `12345`, with a 16-byte salt `S`:

```
honest record
  P || S                                || "12345"
    └─ 16 bytes ─┘                    └─ 5 bytes ─┘

the same bytes, re-presented
  P || S || "12"                        || "345"
    └─ 18 bytes ─┘                    └─ 3 bytes ─┘

the concatenation is byte-identical → identical leaf
                                     → identical Merkle root
                                     → identical record commitment
```

The second presentation passes a validator that accepts "at least 16 bytes" of salt. Verification returns **ok** on a record whose value has been changed from twelve thousand three hundred and forty-five to three hundred and forty-five — defeating the very check written to catch an edited field, and doing it while producing a commitment identical to the one that was signed.

The repair is two independent changes, and it is worth being explicit about why both, since either alone appears sufficient:

- **Length-prefix every component.** With `u32be(len)` before each field, the boundaries are recoverable from the hashed bytes; moving a byte from the value into the salt now changes two length words and the digest with them.
- **Fix the salt at exactly 16 bytes.** A variable-length secret component adjacent to attacker-chosen data is the ingredient that made the attack possible at all.

Either fix closes this specific attack. Both are applied because a construction that depends on a single property being upheld is one refactor away from failing, and because "at least" in a length check is a pattern that reappears. Defence here is cheap; the failure is silent.

THE GENERALISABLE LESSON

Any time a hash is taken over a concatenation of variable-length components, the encoding must be *injective* — distinct inputs must produce distinct byte strings. Concatenation alone is not injective, and the bug it produces is invisible in tests written from the happy path, because the honest encoder never generates the ambiguous case. This is the same failure class as length-extension confusion and as the canonicalisation bugs that have appeared in signature schemes for decades; it is simply easy to reintroduce whenever someone writes `a + b + c`.

§ 5

Withholding, and why the label needs its own commitment

To withhold a field, the discloser publishes the field's *leaf hash* and drops the value and the salt together. The root still recomputes, so every other field verifies untouched, and — crucially — the withholding is visible as a **withholding** rather than as an absence. A recipient can see that a field exists, what it is called, and that it was not shown.

That last clause hides a subtlety that took a second review to surface. The path *is* committed, inside the leaf. But a verifier holding a withheld field has neither the value nor the salt, and so cannot recompute that leaf — which means it cannot check the path either. The label is therefore unverified, and a discloser can relabel it:

```
what the recipient is shown
auditRef      withheld    leaf = 8b5652140e0a0c83...

what was actually committed
contractRef   withheld    leaf = 8b5652140e0a0c83... ← same leaf
```

The verifier confirms the record — and in doing so confirms a false answer to the one question a redacted record exists to answer: *what was withheld?* "A routine audit reference" and "a commercial contract reference" are different disclosures, and the construction as described cannot tell them apart.

The fix is a second, separate commitment carried by every withheld field:

```
pathCommitment(path) = SHA-256( 0x03 || u32be(len P) || P )
```

Distinct prefix, same length discipline, and independently checkable without the salt. A verifier now confirms the label of a field whose value it cannot see, which is exactly the right amount of knowledge to give it.

§ 6

The second tree: over records, doing a different job

The tree in §3 protects the *inside* of one record. A second tree, built over records rather than fields, protects a record's *place in history* – and conflating the two is a reliable source of confusion.

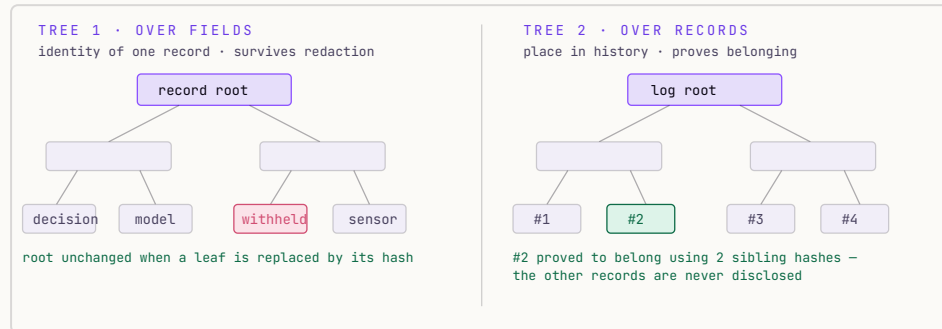


FIGURE 1 Two trees, two jobs. The first makes a record checkable after a field is withheld. The second makes a record provable as a member of a log without producing the log.

Over the second tree, two proofs matter, and they answer different questions:

- **Inclusion.** Given a record and roughly $\log_2(n)$ sibling hashes, anyone can recompute the published root. This proves the record belongs to the log *without revealing any other record*. For disclosure that is the whole game: the volume you hand over is set by relevance rather than by the shape of your storage, and it grows with the logarithm rather than the count.
- **Consistency.** Given an earlier root and the current one, a proof that the earlier tree is a *prefix* of the current one – that records were appended and none of the earlier ones were edited, reordered or removed between two observations. This is what turns "here is a root" into "here is a root that still contains the one you were given last quarter".

§ 7

Two details of RFC 6962 that implementations get wrong

Both trees use the Certificate Transparency construction rather than a hand-rolled one, and there are two places where a plausible-looking implementation is subtly different – both of which have caused real vulnerabilities.

Domain separation between leaves and nodes

A leaf is `SHA-256(0x00 || data)`; an interior node is `SHA-256(0x01 || left || right)`. Without those prefixes, a value that is an interior node in one reading can be presented as a leaf in another, and two different record sets can be made to share a root. The prefixes cost one byte and remove the entire class.

Odd nodes are promoted, never duplicated

When a level has an odd number of nodes, the common shortcut is to duplicate the last one so it can be paired. RFC 6962 does not do this: it splits at the largest power of two *strictly less than* n , which promotes the unpaired node up a level unchanged. The difference is not cosmetic. Duplicating the final node makes distinct trees collide — a tree of n leaves can be made to produce the same root as a tree with the last leaf repeated. This is the defect catalogued as CVE-2012-2459, and it is the reason a tree implementation should be checked against the RFC rather than written from the diagram.

One further value worth pinning: the root of the *empty* tree is SHA-256 of the empty string. Implementations disagree here more often than anywhere else, usually by returning zero or by treating it as undefined, and it is the first vector any cross-implementation corpus should contain.

§ 8

What is deliberately not decided here

The construction commits to *top-level fields*. That is the shallowest granularity that satisfies the requirement, and it is a choice rather than a necessity: a deeper one — a leaf per nested path, or per array element — is equally implementable and gives finer redaction at the cost of a larger disclosure and a more complex proof.

Which granularity is right depends on what records contain, and what records contain is a product decision, not a property of the construction. These notes deliberately stop at the boundary. Likewise absent: any opinion about field naming, event taxonomy or classification. A construction that is indifferent to its payload can be reused; one that encodes a schema cannot.

ON THE PROVENANCE OF THIS MATERIAL

The construction described here was developed inside a commercial evidence system, and the defect in §4 was found by adversarial review of that system before anything consumed it. It is published because a construction that only its authors have attacked is a construction that has not been attacked. The specific product internals — what a record contains, how it is stored, how it is served — are out of scope, as set out in §8.

References

- 1 Directive (EU) 2024/2853, OJ L, 2024/2853, 18.11.2024, Article 9(3) (disclosure limited to what is necessary and proportionate) and Article 9(4)–(5) (protection of confidential information and trade secrets). Cited for context; not legal analysis.
- 2 B. Laurie, A. Langley, E. Kasper, **Certificate Transparency**, RFC 6962, June 2013, §2.1 (Merkle Tree Hash, leaf and node prefixes, the split at the largest power of two strictly less than n) and §2.1.1–2.1.2 (audit and consistency proofs).
- 3 CVE-2012-2459 — Merkle tree collision arising from duplication of an unpaired final node.
- 4 NIST, **Secure Hash Standard (SHS)**, FIPS PUB 180-4, August 2015.

A signature that knows *where it lives.*

Most signed-record designs choose what to sign by asking what is convenient. The better method is to enumerate the replay attacks a signature must survive and let each one **add exactly one field** to the signed bytes. Done that way the preimage is short, every component is justified, and the omissions become visible — including the omission that let a complete, correctly-signed history be presented as a different organisation's.

PAPER · 003 PUBLISHED · 31 AUG 2026 READING · 12 MIN
LICENCE · CC BY 4.0

Canonical · digital1.foundation/articles/the-project-handshake/003-a-signature-that-knows-where-it-lives.html

§1

Derivation, not decoration

A record has been given an identity by [Paper 002](#): a Merkle root over its salted fields, stable under redaction. Signing that root proves an author. The question is what *else* the signature should cover, and the honest way to answer it is adversarially: start with the minimum, find the attack, add the field that closes it, repeat.

What follows is that derivation in order. Each step is an attack that works against the previous step's construction.

§2

Step 0 — sign the record hash

```
signed = recordHash
```

Attack. A signature is a portable object. Lift it, together with its record, into an entirely different set of records. Nothing in the signed bytes says which collection this record belonged to, so it verifies wherever it is placed. A record of a routine decision can be inserted into a history that never contained it.

§3

Step 1 — bind each record to the one before it

Give each record a link to its predecessor — `link = SHA-256(prevLink || recordHash)` — and sign that. Now records form a chain, and an edit anywhere breaks every link after it.

```
signed = SHA-256(prevLink || recordHash)
```

Attack. Chains repair. An adversary who edits record 4 recomputes record 4's hash, then recomputes every downstream link, and produces a chain that is internally flawless. If they also hold the signing key the chain is indistinguishable from an honest one. Even without the key, the structure gives no reason a signature valid at one position should not be valid at another where the surrounding hashes happen to line up.

WORTH STATING PLAINLY

Hash-linking makes tampering *expensive and downstream-visible*. It does not make it detectable on its own. Every additional field below exists because the chain alone is not enough, and a design that stops here — many do — is weaker than its diagram suggests.

§4

Step 2 — bind the position

Include the sequence number in the signed bytes. A signature now attests *this record, at this position*, and cannot be moved even if the surrounding hashes are arranged to accommodate it.

One design note that matters more than it looks. Sequence numbers must be required to **strictly increase**, and must **not** be required to be consecutive. A verifier that demands contiguity will report intact chains as tampered the moment the counter behind the sequence is shared across tenants — two parties appending concurrently legitimately hold 1, 3, 5 and 2, 4, 6. This is not hypothetical: it was shipped, and it produced false alarms on correct data, which is the failure mode most corrosive to trust in a verification tool. Nothing is lost by relaxing it: a dropped record is caught by the link, and a reordered one by its own signature.

§5

Step 3 — bind the domain

Attack. Two different systems adopt this same construction — likely, since it is built from public standards. A record signed by one now replays as a record of the other, because the signed bytes describe a position in a chain and say nothing about *which system's* chain. Worse if the two share any key material or any verifier.

The fix is a domain-separation tag: a constant string identifying the construction and the system, included in the signed bytes. The important design rule is that it must have **no default**. A tag that falls back to a shared value when unset defeats the separation exactly when someone forgets to set it — which is the only circumstance in which it matters.

§ 6

Step 4 — bind the owner (the one that was missed)

Everything above binds a record to a content, a position and a construction. None of it binds a record to a *party*. In a multi-tenant system that omission is severe, and it is the one that shipped.

DEFECT · FOUND BY ADVERSARIAL REVIEW, 31 AUGUST 2026

A complete, correctly-signed history can be relabelled as another party's

The conditions are ordinary rather than exotic. One signing key per deployment, rather than per tenant — a normal choice. A sequence counter shared across tenants — also normal. A preimage of `domain || seq || prevHash || recordHash`, containing no tenant identifier.

Under those conditions, relabelling every row of one organisation's chain to another organisation leaves *an intact, fully-signed, fully-verifying chain attributed to the wrong party*. No forgery is required. No key is compromised. Every hash recomputes, every link holds, every signature checks — and the verifier reports success, because the question "whose is this?" was never asked of the signature.

For a system whose entire claim is *this party did this*, attribution is the one property the signature must carry. It was the one property it did not.

The closed form adds the chain identifier — the identity of the party the chain belongs to:

```
preimage = domain | chainId | seq | prevHash | recordHash

example, as UTF-8 bytes
example.evidence.1.link|did:example:org:northwind|2|c06cbe0d...|a0a252e4...
```

Two rules attach to that layout, and both are the kind that get dropped in a rewrite:

- Neither the domain nor the chain identifier may be empty, and neither may default. A verifier asked to check signatures without being told which chain it is checking should refuse, not guess. Refusing is the correct answer: the alternative is a tool that reports a lifted chain as valid.
- Neither may contain the separator. Otherwise a value can be smuggled across a field boundary — a chain identifier ending in `|9` produces the same preimage as a different identifier with sequence 9. This is the same injectivity requirement as Paper 002 §4, arriving in a different costume. Field-separated encodings need the separator excluded from the fields; length-prefixed encodings do not, and are the safer default when the fields are not under your control.

§7

Migrating a signature format without re-signing anything

The defect in §6 raises an awkward question: how do you change what a signature covers when records are, by construction, append-only? You cannot re-sign old records — rewriting them is precisely the operation the system exists to detect, and a migration that touches historical rows destroys the property it was meant to protect.

The answer is to make the envelope a per-record field rather than a system-wide setting:

- 1 Every record carries the version of the format it was written under.
- 2 The **writer** emits only the current version.
- 3 The **reader** understands every version, dispatching on the record's own field, and chooses the correct preimage layout per record.
- 4 The verdict reports which properties actually hold. A chain containing pre-migration records is reported as verified **and not tenant-bound** — two separate facts, neither hidden.
- 5 Older versions are accepted only behind an explicit flag, so that reading legacy records is a deliberate act rather than a silent default.

No record is ever rewritten. The chain migrates at a boundary, and the boundary is visible to anyone examining it. The cost is that the old format is now permanent — a reader must understand it forever — which is a strong argument for getting the preimage right before the first record is written, and the reason this paper exists.

§ 8

Why Ed25519

The signature scheme is chosen for boring reasons, which are the correct reasons.

- **Deterministic.** Ed25519 derives its per-signature nonce from the key and the message.^[1] There is no random nonce, and therefore no nonce-reuse failure mode — the class of defect that has repeatedly leaked ECDSA private keys, including in widely-deployed production systems. Removing a failure mode entirely is worth more than mitigating it.
- **Independently implemented and verifiable with stock tooling.** A sceptic can check a signature with common command-line tools and no software of ours. That is requirement 2 of Paper 001, and it eliminates schemes whose only implementation is the vendor's.
- **Small and fast enough to be checked in a browser,** which matters because the cheapest way to make verification credible is to let someone do it in front of you without installing anything.

The hash is SHA-256^[2] for the same reason: an examining party's expert can recompute it with a standard utility and move on. Novel primitives are a liability in a context where the goal is for the other side to stop arguing.

§ 9

Two failures that verifiers routinely treat as non-failures

The construction is only as good as the code that checks it, and two specific mistakes appear repeatedly. Both were found in working systems; both are one operator in a boolean expression.

DEFECT CLASS · OBSERVED IN MULTIPLE INDEPENDENT IMPLEMENTATIONS

A missing signature treated as nothing to check

```
the defective guard
if (publicKey && entry.signature && !verify(publicKey, data,
entry.signature)) {
    fail()
}
```

Read it as an adversary. If `entry.signature` is absent the condition is false, the check is skipped, and verification succeeds. So the attack is not to forge a signature — it is to **delete it**. Rewrite the payloads, recompute the hashes and links, remove the signatures entirely, and the verifier reports an intact chain.

The rule: when a public key has been supplied, ***a missing signature is a failure, never a skip***. The presence of a key is the caller stating that signatures are required.

DEFECT · FOUND ON THE BROWSER PORT OF A VERIFIER, 31 AUGUST 2026

A key that will not load, treated as a key that was not supplied

The same defect, one layer up, and considerably harder to see. A key-import function returns `null` on failure — a malformed key, the wrong algorithm, or a runtime that lacks the signature scheme. Callers then write the natural thing:

```
verifyChain(records, { publicKey: key || undefined, ... })
```

`null` is falsy, so the verifier concludes that no key was supplied, skips signature checking entirely, and returns success. Every signature-dependent property silently evaporates — including, in the case where this was found, the cross-tenant check from §6, which is the ***only*** protection a signature provides against relabelling.

The repair is to make the failure representable. Return a distinct sentinel that is **truthy** — so no caller can accidentally demote it to "absent" — and that verifies nothing, so the outcome is an invalid-signature verdict rather than a silent pass. "I was given no key" and "I was given a key and could not load it" are different questions and must not share an answer.

THE PATTERN BEHIND BOTH

Each defect converts a ***required*** check into an ***optional*** one by way of a falsy value, and in both cases the resulting behaviour is indistinguishable from correct operation on honest data. Tests written from the happy path pass. The general rule worth extracting: in verification code, every branch that ***skips*** a check should be justified by an explicit statement of intent from the caller, never inferred from the absence of data — because absence is exactly what an adversary controls.

The result

Five fields, each traceable to an attack that works without it: the construction and system (*domain*), the party (*chain identifier*), the position (*sequence*), the history (*previous hash*) and the content (*record hash*). Deterministic signatures over an injective encoding, an envelope version per record so the format can move without rewriting history, and a verifier that fails closed on every absence.

What none of it does is detect the records that were never handed to the verifier at all. A chain with its final entries removed is a chain that verifies perfectly — every hash, every link, every signature — because verification can only examine what it was given. That is the subject of the next paper, and it is the limit that most systems, including careful ones, describe least honestly.

References

- 1 S. Josefsson, I. Liusvaara, *Edwards-Curve Digital Signature Algorithm (EdDSA)*, RFC 8032, January 2017. §5.1 (Ed25519), including deterministic nonce derivation.
- 2 NIST, *Secure Hash Standard (SHS)*, FIPS PUB 180-4, August 2015.
- 3 B. Laurie, A. Langley, E. Kasper, *Certificate Transparency*, RFC 6962, June 2013 — the tree construction the record identity in Paper 002 is built on.

What verification *cannot* see.

Every construction in the previous three papers checks something that was handed to it. None of them can say anything about what was not. **A chain with its last records removed verifies perfectly** — every hash, every link, every signature — and that is not a defect to be patched but a boundary to be understood, because what lies beyond it needs a party other than you.

PAPER • 004 PUBLISHED • 31 AUG 2026 READING • 11 MIN
LICENCE • CC BY 4.0

Canonical • digital1.foundation/articles/the-project-handshake/004-what-verification-cannot-see.html

§1

The prefix problem

Take an honest chain of twelve records, built exactly as Paper 003 describes. Delete the last two. Hand over the remaining ten.

Every record recomputes to its recorded hash. Every link matches its predecessor. Every signature is genuine — they were never forged, merely not included. A verifier examines the ten records it was given, finds nothing wrong, and reports an intact chain. It is right: those ten records **are** an intact chain. They are simply an intact chain of a shorter history than the one that existed.

THE GENERAL FORM

A valid prefix of a valid chain is itself a valid chain. Nothing inside a set of records can state how many records there should have been, because any such statement is itself a record and can be removed with the others. Self-reference cannot escape this. Verification examines what it is given; the one attack it structurally cannot see is omission.

This is worth dwelling on because it is the limit most often glossed. A system can be described accurately as detecting edits, reordering, insertion and forgery, and still be silent about the simplest attack of all: producing less.

§2

The commitment, and why it must be held elsewhere

The escape is to move one small piece of information outside the system. After writing record *n*, publish a commitment to the state of the chain at that moment — the position, the count, and the link value that record produced:

```
commitment = { seq: 12, count: 12, link: 262d5c40d754777f... }
```

An examining party who obtained that commitment earlier can now detect the truncation: ten records do not satisfy a commitment covering twelve. The commitment is small enough to write in a file note, mail to an auditor, or print.

Two properties of it matter more than the format:

- **It must be obtained earlier.** A commitment handed over at the same time as the records is worthless — the party who trimmed the records would trim the commitment to match. Its entire value is that it was fixed before the dispute existed.
- **It must be held by someone else.** A commitment stored beside the records it protects is protected by nothing. If both live in the same store, the adversary of Paper 001 reaches both.

That second requirement is the crack through which everything in §5 enters, and it is the reason this paper does not end here.

§3

A commitment cannot validate a slice

Here the design admits a subtle and consequential error, which is worth setting out because it looks like a feature.

Disclosure is usually partial in *two* dimensions. Paper 002 handled one — withholding fields within a record. The other is range: a party is asked for last quarter's records, not all of them. That is legitimate and it is the normal case. To verify a range on its own, a recipient needs the link value that the record immediately before the range produced — hand it over and records 400 to 450 can be checked without ever seeing records 1 to 399.

Now combine that with the commitment from §2, and the trap opens.

DEFECT • FOUND BY ADVERSARIAL REVIEW, 31 AUGUST 2026

A range disclosure accepted against a commitment covering a different range

An auditor is given a commitment pinning records 1–10. Later, a discloser produces records 5–14, together with the link value preceding record 5 — and the auditor's own commitment.

```
the auditor holds      commitment { seq 10, count 10, link 4b352aa9... }
the discloser produces records 5..14 + priorLink of record 4
```

```
records 1-4 are silently absent
every signature valid • every link intact • verdict: VERIFIED, exit 0
```

The permissive check asked only whether the committed sequence number appeared *somewhere* in the disclosed set, and whether at least as many records were present as the commitment covered. Both were satisfied. Four records at the front of the history vanished and the tool that exists to catch exactly that reported success.

The repair is to recognise the category error. A head commitment describes a *whole chain*, from origin to a pinned end. It cannot validate a slice, and any attempt to stretch it to one produces a check that passes on suppressed history. So: when a range is being verified — signalled by the presence of a prior link — a whole-chain commitment must be **refused outright** rather than applied loosely. And when a whole chain is verified, the count must match *exactly*, and the committed record must be the *last* one, not merely present.

The right instrument for a slice is a Merkle inclusion proof against a published root — Paper 002 §6 — which proves membership without any claim about the boundaries of what was disclosed.

WHY THE TESTS MISSED IT

The parity suite covering this verifier was thorough and it did not catch this. Its fixtures always set the prior link to null — the whole-chain case — and its only truncation scenario removed records from the *tail*. The suppressed-*front* case was never constructed, so the two implementations disagreed while every test passed. Test suites inherit the imagination of their author; adversarial review exists because that imagination has a shape.

§ 4

"Why not a blockchain?"

The question arrives in every discussion of this material, and it deserves a real answer rather than a dismissal, because the instinct behind it is correct.

What the question is reaching for is a structure where history cannot be quietly rewritten and where no single party decides what the history was. That is a legitimate requirement. But a distributed ledger achieves it by an expensive route — replicating all data to mutually distrusting parties and running consensus to agree an order — and the properties actually needed here are obtainable from a much older and smaller construction.

That construction is an **append-only transparency log**: the RFC 6962 tree of Paper 002, with its two proofs.^[1] No consensus, no token, no network that must outlive the evidence, and — importantly for records that are commercially sensitive — ***no requirement to publish the data itself***, only roots.

PROPERTY WANTED	HOW A TRANSPARENCY LOG PROVIDES IT
This record is in the log	An inclusion proof: about $\log_2(n)$ sibling hashes recompute the published root, revealing no other record.
History was not rewritten	A consistency proof between two published roots: the earlier tree is a prefix of the later one. Records were appended; none of the earlier ones were edited, reordered or removed.
Not one party's say-so	<i>This is the part a log does not give you by itself.</i> See §5.

So the honest answer to the question is neither "we use a blockchain" nor "you don't need one". It is: two of the three properties come from a 1979 data structure standardised in 2013 and deployed in every browser's certificate ecosystem; the third needs a witness, and a distributed ledger is one — expensive — way of hiring several.

§ 5

The gap that remains: split view

Everything so far is checkable by a recipient against material the operator provides. That leaves one attack untouched, and it is the honest limit of a self-hosted log:

THE SPLIT-VIEW ATTACK

An operator maintains **two** histories and shows a different one to each party. Each is internally perfect. Each satisfies every proof, because the proofs are computed against the root that party was given. A recipient checking their own copy finds nothing wrong, because **nothing is wrong with their copy** — the fraud lies in the existence of the other.

No amount of internal cryptography closes this, and it is worth being blunt: a log that is published, verified and audited entirely by its operator has not eliminated the need to trust the operator; it has made that trust precise and narrow, which is progress, but it has not removed it.

Closing it requires a party other than the operator to observe a root and be able to testify to it later. Two mechanisms do this, and they are complementary rather than alternative:

- **Gossip between recipients.** If two parties who were shown different roots ever compare them, the split is exposed by a failing consistency proof. This costs nothing to enable and works only if recipients actually compare — which, in practice, they rarely do.
- **A witness who countersigns.** A third party that observes each published root and signs a statement to that effect. Now producing a second history requires the witness's participation, which moves the attack from "quietly maintain two databases" to "conspire with an external party who has no reason to help you".

§6

Time is a separate problem, and it needs the same kind of witness

A signature proves authorship. It says nothing about **when** — the signer chooses the clock, and the adversary of Paper 001 has the key. Establishing time therefore has the same structure as establishing a single view: it requires someone with no stake in the answer.

The standard mechanism is a timestamp token over a digest, obtained from a timestamping authority under RFC 3161.^[2] The authority signs a statement binding a hash to a time, without ever seeing the data behind the hash. Applied to a published root, it converts "the operator asserts this history existed on Tuesday" into "an independent authority attested to this root on Tuesday" — the same witness function as §5, obtained from an established supply of them.

Within the European framework, this is where a distinction is worth stating precisely, because it is routinely blurred in marketing material. Regulation (EU) No 910/2014 as amended treats electronic timestamps at two levels: ^[3]

PROVISION	EFFECT
Article 41(1) any electronic timestamp	Shall not be denied legal effect and admissibility as evidence in legal proceedings solely on the grounds that it is in electronic form or does not meet the requirements of a qualified timestamp. This removes one ground of objection. It is not a positive claim about anything.
Article 41(2) a qualified timestamp	Enjoys the presumption of the accuracy of the date and time it indicates, and of the integrity of the data to which they are bound. This is a materially different thing, and it is the one worth having.
Article 42(1)	What makes a timestamp qualified: it binds date and time so as to reasonably preclude undetectable change, rests on an accurate time source linked to UTC, and is signed by the qualified provider.
Articles 21(3), 22	Qualified status is constituted by listing on a Member State trusted list. It is therefore per-service and time-dependent: what matters for an archived token is whether the service was qualified at the moment of issuance.

An unqualified timestamp — from a free public authority, say — obtains Article 41(1) and nothing beyond it. It is genuinely useful as an engineering control and it is not the legal instrument that 41(2) describes. Describing a system as offering "eIDAS-compliant timestamping" without saying which of those two it means is, at best, imprecise.

ONE BOUNDARY WORTH KEEPING

A verifier should bind a timestamp token to the digest it claims to cover, and should **not** attempt to validate the authority's own certificate chain. That validation belongs to standard tooling run against trust roots the reader chooses. A verifier that also vouches for its own trust anchors is asking to be trusted twice — which is the failure the whole series is written against.

§7

An honest ledger of what is open

Collecting the limits from all four papers in one place, because a reader deciding whether to rely on this class of construction is entitled to them together rather than scattered:

- 1 Omission is invisible without an external commitment, and the commitment is only worth what the independence of its holder is worth.

- 2 Split view is not closed by internal cryptography. It needs a witness, gossip between recipients, or both.
- 3 Time is not established by a signature. It needs an independent authority, and the legal weight of that authority varies by whether it is qualified in the relevant sense.
- 4 None of this speaks to truth. A verified record of a false statement remains a false statement, verified.
- 5 A construction is only as good as its verifier, and verifiers fail in ways constructions do not — as Paper 003 §9 and §3 above both show, and as anyone shipping one should assume of their own.

None of these are reasons not to build the thing. They are the reasons to describe it accurately: a party evaluating evidence infrastructure is, by disposition, someone who will find the boundary eventually. It is materially better for them to find it in your documentation than in cross-examination.

THE SERIES, IN ONE LINE

Assume the store is lost; give each record an identity that survives redaction; bind every signature to content, position and owner; and then be precise about the fact that what remains still needs somebody else. That last clause is the handshake — the point at which a construction stops asserting and starts being checkable.

References

- 1 B. Laurie, A. Langley, E. Kasper, *Certificate Transparency*, RFC 6962, June 2013, §2.1.1 (audit proofs) and §2.1.2 (consistency proofs). See also the split-view discussion in the Certificate Transparency literature on gossip and witnessing.
- 2 C. Adams, P. Cain, D. Pinkas, R. Zuccherato, *Internet X.509 Public Key Infrastructure Time-Stamp Protocol (TSP)*, RFC 3161, August 2001; updated by RFC 5816.
- 3 Regulation (EU) No 910/2014 on electronic identification and trust services, as amended by Regulation (EU) 2024/1183. Articles 41(1), 41(2), 42(1), 21(3) and 22. Note that Article 41(3) was deleted by the 2024 amendment and mutual recognition now sits at Article 24a(6). Cited for context; not legal analysis.
- 4 R. C. Merkle, *A Certified Digital Signature*, CRYPTO '89 — the tree construction, whose original description dates from 1979.

Three questions, *one substrate*.

Governance, liability evidence and agent authority are routinely sold as one problem. They are three, they face different adversaries in different directions in time, and a system that merges them ends up holding **the union of everyone's data and the intersection of everyone's guarantees**. This paper sets out the layering, what the layers legitimately share, how they compose over a single event – and the seams that are still open.

PAPER · 005 PUBLISHED · 1 SEP 2026 READING · 13 MIN

LICENCE · CC BY 4.0

Canonical · digital1.foundation/articles/the-project-handshake/005-three-questions-one-substrate.html

\$1

Three questions that look like one

An organisation runs an automated system. Something happens. Over the following months three different parties arrive, and it is tempting to think they want the same file.

THE QUESTION	ASKED BY	ITS SHAPE
Are we permitted to do this?	a regulator, an auditor, your own board	Prospective . Answered before and during operation, from the system's role, its risk classification and what triggers apply. The artefact is a <i>derivation</i> : an obligation, with the reasoning that produced it.
What actually happened?	a claimant, opposing counsel, an insurer	Retrospective and adversarial . Answered years later, to someone with an interest in the answer being different, and answered <i>partially</i> – relevance is scoped and confidentiality is protected. The artefact is a <i>record</i> .
Who authorised this action?	a counterparty, at the moment of acting	Instantaneous . Answered in the moment before something irreversible happens – a payment, a contract, a deletion. The artefact is a <i>credential</i> , and its most important property is that it can be withdrawn.

Different time direction. Different adversary. Different artefact. Different retention horizon — an obligation derivation is worth little once the rule changes, whereas a record may need to be checkable for a decade or more.^[1] And, decisively, different **disclosure surfaces**: the material you hand a regulator, the material you disclose to a claimant, and the material you show a counterparty at machine speed have almost nothing in common except the event they describe.

§2

Why they have to be separate layers

The instinct is to build one system that answers all three, because they are about the same event and share vocabulary. Three arguments against, in ascending order of how much they cost you if ignored.

Data gravity works the wrong way

Merge the layers and each one's data becomes reachable from every other one's interface. The authority layer, which needs to hold almost nothing, is now colocated with the record layer, which holds the most sensitive operational history you have. A disclosure order scoped to one produces an argument about the other. Separation is not tidiness; it is what keeps the blast radius of a disclosure, a breach or a subpoena bounded.

The guarantees do not compose upward

Each layer can make a strong, narrow promise. The authority layer can promise that a credential was valid and unrevoked at an instant. The record layer can promise that a record is exactly what was written and has not been quietly altered. The governance layer can promise that an obligation follows from a stated derivation. Merge them and you cannot promise the union — you end up promising the weakest of the three about everything, because a single verifier now has to be right about all of it at once.

Severability is a real requirement, not a business preference

These layers face different regulators, will be operated under different jurisdictions, and have genuinely different lifespans. A system that cannot be separated later has made an irreversible decision on behalf of people who were not in the room. The engineering expression of that is unglamorous and absolute: **separate stores, separate keys, separate verifiers**. Not separate schemas in one database — separate databases, so that carving one out is a move rather than a migration.

THE TEST THAT KEEPS THIS HONEST

For each layer, ask: ***could this be handed to a different operator tomorrow, with its records still verifying, without untangling anything?*** If the answer is no, the layers are merged whatever the architecture diagram says.

§3

What they share, and why sharing is safe

Separate stores, separate keys — and yet all three need the same underlying property: a record that proves itself, in an order that cannot be quietly rearranged, attributable to a party. Rebuilding that three times is how you get three subtly different constructions and three different sets of defects. (That is not hypothetical. The audit behind these papers found the same canonicalisation defect — Paper 002's `__proto__` trap — in eight copies across four codebases, fixed in some and live in others.)

So the layers share a **substrate**, which is the set of constructions in Papers 002 and 003:

- canonical JSON — deterministic, keys sorted recursively, null-prototype accumulator;
- a record identity that is a Merkle root over salted field leaves, so it survives redaction;
- a hash link binding each record to its predecessor;
- a signature over `domain || chainId || seq || prevHash || recordHash` ;
- a per-record envelope version, so the format can move without rewriting history;
- and the rule that every absence — a missing signature, an unloadable key — is a failure.

THE DISTINCTION THAT MAKES THIS WORK

The substrate is shared by **specification**, not by deployment. Each layer keeps its own store, its own signing key, and — critically — ***its own domain tag***. They implement the same construction; they do not share a database, and they do not share a namespace.

That last point is what makes sharing safe rather than dangerous. Domain separation (Paper 003 §5) means a record signed by one layer **cannot replay as a record of another**, even though the byte layout is identical and even if a key were ever reused. The tag is what converts "three systems using the same maths" from a correlated-failure risk into a

straightforward benefit: one construction to get right, one verifier shape to review, one set of test vectors — and no path by which an artefact of one becomes an artefact of another.

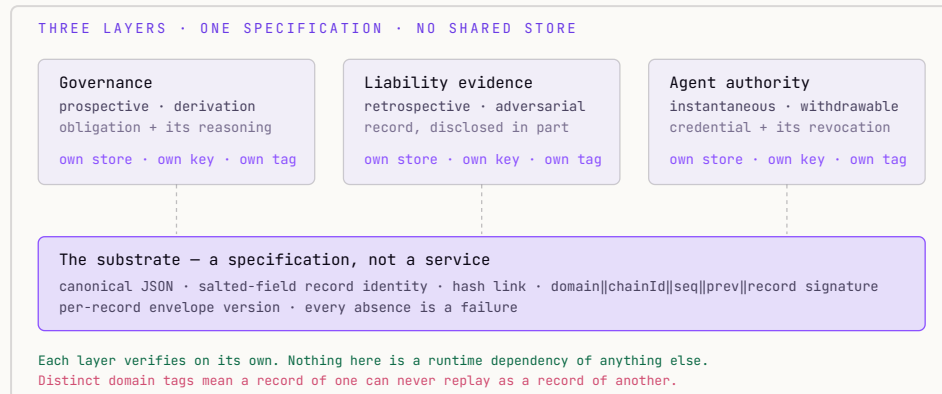


FIGURE 1 The layering. The substrate is drawn beneath because every layer implements it — not because anything runs there. There is no shared service, no shared database and no runtime coupling; the only thing held in common is the specification.

§4

How they compose over one event

The value of the layering shows up when a single event is examined from three directions at once. Take a concrete shape: an automated system takes an action with financial consequences, and eighteen months later somebody disputes it.

- 1 The **authority layer** answers what was permitted *at the instant*: which agent identity acted, under what scope, within which ceiling, and whether that authority had been withdrawn. Its record is small and deliberately says nothing about amounts or people.
- 2 The **evidence layer** answers what the system actually did and what state it was in — disclosed in part, with fields withheld for relevance or confidentiality, and the remainder still reproducing the commitment sealed at the time.
- 3 The **governance layer** answers which obligations attached to that system, on what derivation, on the date in question — including obligations that have since changed, which is why the derivation matters more than the conclusion.

Three artefacts, three narrow claims, **each checkable without the other two**. That independence is the whole point: a party who trusts none of them can verify each separately, and a failure in one does not contaminate the others. A merged system produces one artefact that must be believed as a whole.

WHAT COMPOSITION DOES NOT GIVE YOU

Three verified artefacts about one event do not add up to a verified account **of** the event. Each says something narrow and true; the joining-up is an argument made by people, from evidence, in a forum. A system that claimed to produce the conclusion would be making exactly the overreach these papers exist to avoid.

§ 5

The seams, including the one that is not finished

Layers that must not share a database still have to meet somewhere. Three seams, and they are not equally solid.

Identity and tenancy

All three need to agree on **which organisation** a record belongs to, or the chain identifier in the signed preimage means different things in different systems. The safe form of this seam is a shared identifier resolved over an API — never a shared table. The identifier appears inside the signed bytes of each layer under that layer's own domain tag, which is what keeps the binding strong without creating a coupling.

A common verifier contract

Because the substrate is one specification, a party can hold one mental model and check all three: recompute the canonical form, recompute the hash, rebuild the link, check the signature over the five-field preimage, and refuse on any absence. Reviewing one verifier carefully means understanding all of them — which is a real reduction in what a sceptic has to take on faith.

Custody hand-off — and this one is open

OPEN PROBLEM · IDENTIFIED 31 AUGUST 2026

A record handed between layers can arrive as trust-on-receipt

Where one layer takes custody of another's records — a governance layer archiving what an enforcement point emitted, say — the natural implementation posts a payload over an authenticated channel. The audit behind these papers found a hand-off whose posted body carried the record content and its hash but **neither the signature nor the envelope version**.

The receiving layer therefore cannot verify what it was handed. It can only trust the channel — so an artefact that was self-proving on one side of the seam becomes an assertion on the other, and

the property is lost exactly at the boundary where two parties are involved and it matters most.

The rule this yields is worth stating generally: **a record must cross a boundary with everything needed to verify it** — signature, envelope version and chain identifier — or it does not cross as a record. Transport authentication tells you who sent a thing; it says nothing about whether the thing is what it claims to be.

§ 6

What the layering costs

An architecture paper that only lists benefits is a diagram with adjectives. The costs here are real and two of them are unsolved.

- **Three key custodies.** Three signing keys, three rotation stories, three sets of published verification keys. Each is a place to get key management wrong.
- **Three domain tags that must never collide,** including as products are renamed, merged or sold. A tag is a permanent commitment; it appears in every signature ever made.
- **Cross-language agreement is not free.** Where layers are implemented in different languages — and they will be — the canonicalisers must agree byte for byte. They do not agree by default. Measured across two runtimes on the same inputs: integers above 2^{53} serialise differently, small exponents format differently (`1e-7` against `1e-07`), and object keys containing supplementary-plane characters sort differently under UTF-16 code units than under UTF-8 bytes. Each produces a different record hash for the same logical payload.
- **There is no shared conformance corpus yet.** This is the honest gap. Agreement is currently believed rather than proven, and the fix is unglamorous: a versioned set of test vectors — canonicalisation, preimages, tree roots, proofs, and named negative cases — that every implementation must reproduce, wired into every build. Until that exists, "these implementations agree" is a claim of the kind this series exists to distrust.

A DECLARED NUMBER POLICY IS NOT OPTIONAL

The cross-language divergences above are not exotic edge cases; they are what happens when a specification omits to say how numbers are written. Any shared substrate must state its number encoding and its key-ordering rule explicitly, and test them — or restrict payloads to a subset where the runtimes cannot disagree. Leaving it implicit means the disagreement surfaces as an unverifiable record, years later, in front of the party least inclined to be understanding.

§7

Provenance, and what is out of scope

The layering described here is implemented across three systems built by the practitioners contributing these notes: a governance layer (***RiverAct***), a liability-evidence layer (***SureWard***) and an agent-authority layer (***SignWard***). That provenance is stated so a reader can weigh it, and because the defects reported throughout this series were found in that code rather than imagined.

What is deliberately absent, here as elsewhere: the content of any layer's rules — the obligation derivations, the detector logic, the policy evaluator — and every operational specific. Those are product decisions and deployment decisions. They vary, they are not properties of the architecture, and generalising from one implementation would be misleading rather than generous.

What is present is the part that is checkable: three questions, three layers, one substrate held in common by specification, distinct domain tags to keep them apart, and an honest ledger of the seams that are not yet finished. Each of those claims can be tested by anyone who implements it, which is the only standard worth writing to.

References

- 1 Retention horizons differ by instrument. Regulation (EU) 2024/1689 sets a floor of at least six months for automatically generated logs (Articles 19(1) and 26(6)); Directive (EU) 2024/2853 sets a ten-year expiry period, extended to twenty-five years for latent personal injury (Article 17). Cited for context; not legal analysis.
- 2 The Project Handshake, Paper 002 — ***Two trees***: the record identity and the salted-leaf construction referenced in §3.
- 3 The Project Handshake, Paper 003 — ***A signature that knows where it lives***: the signed preimage, domain separation and envelope versioning referenced in §3 and §5.
- 4 The Project Handshake, Paper 004 — ***What verification cannot see***: the limits that apply to every layer here, including the witness gap that none of them closes alone.

What a revocation *cannot recall.*

Revocation is almost always specified as a state change — a bit flips, a list is reissued, a status reads *revoked* — and almost never as what it is, which is a claim about a set of actions. That set is not a property of the mechanism at all. It belongs to **where** the checks are, what they do when they cannot read the status, and whether anything still holds the action.

PAPER • 006 PUBLISHED • 1 SEP 2026 READING • 15 MIN

LICENCE • CC BY 4.0

Canonical • digital1.foundation/articles/the-project-handshake/006-what-a-revocation-cannot-recall.html

§1

The question people ask, and the question they mean

Every discussion of revocation reaches the same question within minutes: *how quickly does it propagate?* It is the question with a number attached, and a number looks like an answer. The question underneath it is about consequences: **what can still happen after the decision that it should not?**

Take two systems whose distribution differs by orders of magnitude. If neither still holds the action when the withdrawal lands, both produce the same outcome and the speed bought nothing. Now take two whose distribution is identical, one re-checking at the last moment it still holds the action, the other only on admission. Those differ enormously. Propagation was constant; the set of actions that could still commit was not.

THE RESTATEMENT THIS PAPER WORKS FROM

A revocation is not a state change with a latency. It is a *claim about a set of actions* — the set that can still commit after it — decided by the enforcement topology rather than by whatever publishes the status. That makes it answerable by construction, which is a better position than answerable by benchmark.

The instruments do not settle it. Regulation (EU) 2024/1689 requires, at Article 14(4)(e), that a provider enable a human overseer – as appropriate and proportionate – to intervene in the operation of a high-risk system or interrupt it through a stop button or a similar procedure allowing a halt in a safe state, with a suspension duty on the deployer at Article 26(5); both apply from 2 December 2027 for Annex III systems and 2 August 2028 for Annex I embedded ones.^[1] The voluntary NIST framework asks, at MANAGE 2.4, for mechanisms to supersede, disengage or deactivate systems performing inconsistently with intended use.^[2] Neither says which actions must not commit, nor where that stops being true.

Paper 005 §1 said of the credential answering *who authorised this action?* that its most important property is that it can be withdrawn, and then went no further. This paper is that follow-on, and it concedes at the outset that the property named there is not a property of the credential. A credential can *express* a withdrawal; what decides its effect is everything below.

§ 2

Blast radius is a specification obligation

A withdrawal has to name what it withdraws, and the naming belongs in the specification rather than in the operator's head at the time. Three radii cover the cases that arise, and they earn their place by differing in kind rather than in size.

RADIUS	CHARACTER	WHAT IT WITHDRAWS, AND WHAT IT LEAVES STANDING
One mandate	narrow, ephemeral	The authority for a single engagement. Every other authority the same actor holds is untouched, which is what makes it the radius for the ordinary case.
One agent class version	wide, durable	The authority of every instance running a particular version. The radius for a defect: the fault is in the build, so the withdrawal follows the build rather than chasing instances.
One organisation	widest, durable	Every authority issued under an organisation. The radius of last resort, and – as §6 argues – the one that most resembles an outage when fired by somebody who should not have been able to fire it.

An unnamed radius is a defect rather than an omission, for reasons that worsen in order. The operator cannot state what they have just done, which is the first thing they will be asked. The record cannot state it either: an entry reading *revoked* with no named extent is not evidence a third party can check. And control cannot be graded, which is §6.

The named set is also only half of it, because the achieved reach is that set intersected with the enforcement points which actually consult the status.

THE TWO HALVES OF AN HONEST RADIUS

Everything under this organisation is a specification, and it is stable. **Everything that passes a point which checks** is an inventory, true on the day it was taken — and systems grow by acquiring paths: a new integration, a direct call added under deadline, a batch job predating the enforcement layer. Merging the two into one sentence turns a decaying inventory into an apparent guarantee, and the merge is rarely noticed, because both halves are true when it is written.

§3

Absence is not health

The substrate of Paper 005 §3 ends on a rule: every absence — a missing signature, an unloadable key — is a failure. Transposed to the live path it becomes the most consequential decision in the design. What does an enforcement point do when it cannot read the status at all?

The failure is undramatic. A point holds a status it fetched earlier, the distribution path stops working, and it now knows what was true earlier and nothing about now. Three behaviours are available, and one of them is a decision.

- **Evaluate on the last status it could read.** Failing open, with a specific hazard: an adversary who wants a withdrawal not to be honoured need not defeat the withdrawal, only make the status unreachable.
- **Evaluate, and record a warning.** Failing open with paperwork. The action commits; the warning is read by whoever investigates why it did.
- **Refuse the actions whose cost of being wrong is unrecoverable.** Not "refuse everything", which turns a status outage into a total outage and will rightly be switched off by the first person it wakes at night — but a decision made on the absence of evidence, which is the only kind available when evidence is what is missing.

the enforcement point asks one question: how old is the status I hold?

within the refresh bound	→ evaluate	a decision made on evidence
past the staleness threshold	→ refuse	a decision made on its absence
status unreadable	→ refuse	the same decision, for the same reason
evaluate anyway, on whatever was last read		← the default that fails open

The cost is better stated than discovered: a fail-closed rule turns an outage in the status path into refused live traffic, and somebody will ask for the switch that disables it. That switch is the defect. What is legitimate is to bound the window in which refusal happens, which is §4.

SHADOW MODE IS ABSENCE OF COVERAGE

Enforcement points commonly ship a shadow mode: evaluate, record what would have happened, allow the action anyway. It is how an operator learns what a rule will do before it does it. It is also a state in which a point **cannot refuse**, and a point that cannot refuse is a logger — so it belongs in the inventory of §2 as absence of coverage, not coverage with a footnote.

§4

The interval, honestly

An interval remains between the moment a withdrawal is decided and the moment every point that checks would see it. Two very different statements can be made about it, and they are routinely offered as the same kind.

A **measurement** describes one deployment, on one day, on a network that was behaving. It binds nothing — including the same system tomorrow — and it degrades in precisely the conditions under which it would be relied upon, since the interesting case for a withdrawal is rarely a quiet Tuesday.

A **bound** is a different object, and its force comes from enforcement rather than observation. "The status list is refreshed every ***T***" is not a bound; it is a hope about a scheduler. The bound is ***no action of this class commits against a status older than T***, enforced at the point that would otherwise commit it. Where a signed push sits above a scheduled refresh, the push is an optimisation and the refresh is the guarantee, because a push never delivered is indistinguishable, at the receiving end, from a push never sent.

THE SENTENCE TO WRITE, AND THE SENTENCE NOT TO

Not: ***a withdrawal reaches enforcement in X***. That is a measurement of one machine on one day, and it will be quoted back long after both have changed. But: ***no action of this class commits against a status older than T, and when the distribution path is unavailable a status is at most T-max old — both configured, enforced and inspectable***. The second sounds weaker and is far more defensible, because a sceptic can check it without the operator's cooperation, which is the standard set in Paper 001.

§5

The commit boundary

Everything so far shortens an interval. None of it addresses what the interval is an interval *of*, and that is settled by one decision: where the last check sits.

A check at admission answers *was this permitted when we began?* A check at commit answers *is this permitted at the last moment we can still decline?* For an action that begins and ends at the same point those are one question. For anything queued, batched, retried, scheduled or waiting on a person they are two, and only the second is about the present.

The rule that follows is short. Place a check at the last boundary at which the system still holds the action, and make it the full check. The qualifier is load-bearing: a re-check that consults a stored verdict from the earlier one is the earlier answer asked again. The defect shape is a flag on the action recording that it was authorised — exactly the artefact a well-meaning optimisation produces, and exactly what converts the one check that could have observed the withdrawal into one that structurally cannot.

One case holds the largest interval in most systems. When an action is parked for human approval the gap runs to minutes and sometimes hours, and it is the gap most likely to overlap a withdrawal, for an unglamorous reason: withdrawals are decided by people too, and people decide during the same working day. Re-running the full check at the moment of approval closes it — an approval granted after a withdrawal does not commit.

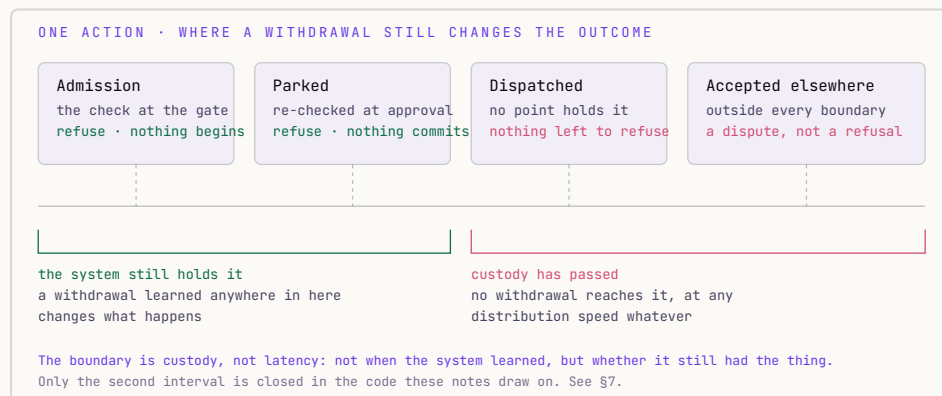


FIGURE 1 The life of one action. Distribution speed moves the moment of learning along the rail; it never moves where the rail divides.

A refusal at any of these points should be written — what was refused, and which authority failed — into the organisation's own signed chain, on the substrate of Paper 005 §3. The dispute that arrives later is rarely *was it withdrawn*; it is *what was still allowed to happen afterwards*.

The widest lever needs two hands

A withdrawal is also something somebody writes, and writing it is a privileged operation whose danger scales with exactly the radius §2 insisted on naming. Turn the organisation-wide radius around: a mechanism that can withdraw every authority in an organisation can halt that organisation's automated operations on the strength of one credential. It is not only a safeguard; it is the most attractive attack surface in the system, because it needs no exploit at all — only the credential, used exactly as designed.

- The narrow radius stays single-control, deliberately rather than as a concession: the urgent ordinary case must not need a committee, and its blast radius is bounded by construction.
- The durable radii require a distinct second approver, and the distinctness is the entire property. A second approval from the same principal, a shared account, or a service credential held by the same process is an audit row rather than a control. It belongs in the specification as an inequality between two identities, not in a runbook as a procedure.
- The durable radii should be appended to a transparency log and return an inclusion proof (Paper 002 §6), because the contested question afterwards is usually *when* — and Paper 004 §6 applies in full: the operator's own timestamp on the operator's own log still wants a witness.

DEFECT • FOUND BY ADVERSARIAL REVIEW, 1 SEPTEMBER 2026

Two levers of comparable width, two control models, one estate

The audit behind these papers examined two mechanisms that can each stop work across an organisation. The authority layer required a distinct second approver for both of its durable radii and appended every such withdrawal to a transparency log. The policy gate, whose standing refusals can deny an entire class of calls, accepted a rule write on one API credential, with an audit row and no second approver at all.

Neither is wrong in isolation; the defect is the inconsistency, and it survives review because each half is examined by the people who built it. An attacker takes the weaker path, so a control model is only as strong as the least-controlled lever of comparable width in the estate — which is the rule: **grade control by blast radius, not by component.**

Dual control adds a step to the action one least wants to slow, and there are two answers. The bad one is a break-glass path around the second approver — a single-credential route to the widest radius, under a name that sounds careful. The good one is the grading itself: the narrow radii exist so that the urgent case is a narrow one. If containing a failure routinely requires the organisation-wide lever, the radii were drawn in the wrong places.

§7

What withdrawal cannot recall

Withdrawal has a floor, and the floor is not latency. It is **custody**. Once an action has left the last enforcement point that held it — a request already dispatched to a downstream tool, a response already streaming, a batch already handed to a provider, an instruction already accepted by a counterparty's system — no withdrawal reaches it. Not a faster one, not a better-distributed one, not one decided a moment after the event rather than a minute after it. The boundary is not *when did we learn*; it is *do we still hold it*. Everything §3 and §4 buy is bought inside that boundary, and none of it buys anything outside.

Exactly one instance of the in-flight case is closed in the code these notes draw on, and it is worth describing precisely, because the temptation is to present it as the answer. The re-verification at approval described in §5 closes the interval between parking and approval — often minutes, sometimes an hour — and for a specific reason rather than a general one: that is the interval a person spends deciding, which is the interval most likely to overlap another person deciding to withdraw. It closes nothing else.

OPEN PROBLEM • IDENTIFIED 1 SEPTEMBER 2026

There is no measured propagation figure, and bounds are not measurements

The second admission this series' standard requires concerns what these notes do not have. There is no measured propagation figure for any of this. What exists is a set of **configured worst-case bounds**: a signed push between five-minute status-list refreshes; a fifteen-minute staleness threshold past which mutating actions are refused; and a maximum status age of five minutes when the push path is unavailable.

Those are bounds — configuration, enforced at the points that would otherwise act, inspectable by somebody who does not trust the operator. A bound deliberately says nothing about how a system usually performs; it says what the system will not do, which

is the half that still holds on a bad day. Closing this needs a harness exercising the distribution path under partition and under load, publishing distributions rather than a figure.

The limits, collected, because a reader deciding whether to rely on this class of control is entitled to them together rather than scattered:

- 1 **Custody is the floor.** Beyond the last point that holds an action, no improvement in distribution reaches it.
- 2 **Coverage is an inventory claim,** shadow mode is absence of coverage, and the inventory decays unless it is deliberately held.
- 3 **One in-flight interval is closed** — parking to approval — because the system had been asked to wait, not because the general problem was solved.
- 4 **Graded control is only as good as the distinctness of the identities it rests on.**
- 5 **Withdrawal ends authority; it does not undo consequences.** What was done under an authority valid at the time remains done, and unwinding it is a matter for people, argued from records — which is why the record layer of Paper 005 sits apart from the authority layer.

None of these is an argument against building the mechanism; a withdrawal that closes the intervals it can close is worth far more than one that closes none. They are arguments for describing it as what it is. The temptation here is to answer a question about consequences with a number about speed, because the number is available and the honest answer is a diagram of where the checks are. The number will be believed, and then tested against an action that had already left.

§ 8

Provenance, and what is out of scope

The mechanisms described here are implemented in two systems built by the practitioners contributing these notes: an agent-authority layer (***SignWard***), which withdraws authority at the three radii of §2, and a policy gate (***Gateward***), which can refuse a call before it reaches a provider. The defect in §6 was found by adversarial review across both, which is the only reason it was found at all — each half looked correct to the people who had built it.

A third system in the same estate is named only to say what it is not. The governance layer of Paper 005 (***RiverAct***) names which duties attach to which system on which date, records the decision and computes the resulting deadline. It sits in no request path and it refuses nothing. Saying

so is the same discipline as the rest of this paper: a layer that cannot intervene must never appear in a sentence about intervention, however convenient the adjacency.

Out of scope, here as throughout the series: the content of any rule, the language rules are written in, the shape of any record, the identifiers involved, and every operational specific. What is present is the part a sceptic can check — that a withdrawal is a claim about a set of actions, that the set is fixed by where the checks are and what they do in the dark, and that underneath it lies a floor made of custody which no mechanism described here crosses.

References

- 1 Regulation (EU) 2024/1689, Article 14(4)(e) on human oversight — including the qualification “as appropriate and proportionate” and the alternative of “a similar procedure”, both load-bearing and both routinely dropped in summary — and Article 26(5) on the deployer’s suspension duty. Application dates follow the amendment of Article 113 by Regulation (EU) 2026/1744. Cited for context; not legal analysis.
- 2 National Institute of Standards and Technology, *Artificial Intelligence Risk Management Framework (AIRMF 1.0)*, NIST AI 100-1, January 2023, MANAGE 2.4. The framework is voluntary: it recommends, and does not require.
- 3 The Project Handshake, Paper 005 — *Three questions, one substrate*: §1, the credential whose most important property is that it can be withdrawn, and §3, the substrate on which a refusal is recorded.
- 4 The Project Handshake, Paper 004 — *What verification cannot see*: the witness and the timestamp, which apply to a withdrawal’s own record as much as to any other, referenced in §6.
- 5 The Project Handshake, Paper 001 — *Assume the store is already lost*: the threat model in which the operator sits inside the adversary, referenced in §4 and §6.

Colophon

A

The record

This document is generated from the published pages, which are the record. Where this file and a page disagree, the page is right and this file is stale. Each source below is named with the SHA-256 of the file this document was built from, so a reader can check that what they are holding is what was published; the generator fetches each live page as it builds and reports any that has moved since.

INDEX	digital1.foundation/articles/the-project-handshake/
001	digital1.foundation/articles/the-project-handshake/001-assume-the-store-is-lost.html sha256 387d600a0f57e3f079925371c597ff8c70320cf13a178b949f19381af89efbc7
002	digital1.foundation/articles/the-project-handshake/002-two-trees.html sha256 6f0376459519e7455b78d39c1c9a9fe59cc1e8d6a0e040552eb7fe781e3981b2
003	digital1.foundation/articles/the-project-handshake/003-a-signature-that-knows-where-it-lives.html sha256 eb1116d165e367f904bbf2b5606737f75abcaf34e63888eafa8ce1442a3e5127
004	digital1.foundation/articles/the-project-handshake/004-what-verification-cannot-see.html sha256 7dee5e1d799a946656430715d8b5193370aa0993816bd1d946113410a6fbba81
005	digital1.foundation/articles/the-project-handshake/005-three-questions-one-substrate.html sha256 7c5c019f382c16e442bb5f8445e8b35ea4da8e3e44600aa2c8b6a27d5a1e256d
006	digital1.foundation/articles/the-project-handshake/006-what-a-revocation-cannot-recall.html sha256 9239a57b96c06e64f47eca0ccab439f2a06c95564732c3630a1aef38171bd66a

B

Licence

Every paper in this document is published under the Creative Commons Attribution 4.0 International licence, creativecommons.org/licenses/by/4.0/. You may copy, redistribute and adapt it for any purpose, including commercially, provided you credit **Stichting Digital One Foundation i.o.** and link to the page cited above.



How it is set

In the six faces the site itself carries and embeds: Fraunces 300 and 600 for titles and the emphasised word, Manrope 400 and 500 for the text, JetBrains Mono 400 for the ledger's own labels, code and running matter. Those six are Latin subsets and do not carry every character the papers use — ¶ ² ↔ — | L J are absent from them. Nothing is drawn in their place and quietly dropped from the text: each is set in a named system face. Every character in every paper was extracted back out of this file and checked before it was written, so the whole document copies, searches and cites as it reads.

Figures are the papers' own inline drawings, re-toned from the site's night ground to this page by a one-to-one swap of the closed house palette, so green still means it holds and rose still means it breaks — and the two are separated in weight of grey as well as in hue, so a photocopy keeps the distinction. The rail down the left of each page is the ledger from the Foundation's homepage; a filled dot marks each numbered section, and the § number sits in the gutter, outside the measure.

The sheet is A4 as the page asks for it, which Chromium rounds to 594.96 × 841.92 pt when it writes the file — 0.1 mm narrower and 0.03 mm taller than the 595.28 × 841.89 pt of the standard. No printer will show it; a document that prints its own hashes should say it anyway.

© MMXXVI Stichting Digital One Foundation i.o.
Amsterdam, the Netherlands

Published under CC BY 4.0.
No public-benefit (ANBI) status is claimed at this time.

digital1.foundation
The Project Handshake