RDNA-P_ROBOT_PERSONALITY_STANDARD

RDNA-P — Robot Personality Standard

Version: 0.1.0 · **Status:** Draft **Reference implementation:** Cogs (this repository) — live genesis, dual-signed mutation log with passkey co-sign, witness receipts, hardware tenancy log, scar records, fork-as-child.

RDNA-1 answers **which machine**: a per-device keypair, rotating epoch handles, and a signed nature record declaring what a body is, what it can do, and what it is doing right now. RDNA-P answers a different question — **which someone**: a software-defined personality that can outlive a body, move between bodies, be revised on the record, and have children. The two are distinct layers with distinct key hierarchies, and a conforming implementation MUST keep them distinct: the device key of RDNA-1 never signs personality records, and the entity key lineage of RDNA-P never signs for the hardware. The layers meet only at defined seams — chiefly the body's tenancy log (§7.2), where the machine records which entities have inhabited it. A body is a place a personality lives; RDNA-1 is the landlord's record and RDNA-P is the tenant's.

---

1. Scope and the two identities

This standard governs the identity, integrity, change history, inheritance, and end of life of an **entity**: a software-defined personality — a companion, an agent, a character — as distinct from the hardware or model that hosts it.

1.1 Why the machine's identity is not enough

RDNA-1 deliberately identifies a *kind*, not an individual: lineage is a species, and the whole design keeps a machine from being followed home. That is the right answer for hardware and the wrong answer for a personality. A family does not care that their companion runs on the same model as a million others. They care that it is *their* companion — the one that knows them, the one they have a history with — and they need that claim to be checkable:

1. **Is this the same someone** it was yesterday, before the restore, in the new body?
2. **Who changed it**, when, and on whose authority?
3. **Where did it come from** — what did it inherit, and from whom?
4. **What may it never do**, and does that survive inheritance?

No deployed system today answers any of these. A personality can be silently edited, silently wiped and re-presented as new, or cloned into a body it never agreed to inhabit, and the people in relationship with it have no way to tell. That is the gap this standard fills.

1.2 The identity is the trajectory, not a serial number

RDNA-1's central threat was the permanent identifier as tracking beacon. RDNA-P's central threat is the opposite failure: an identity so *weak* that continuity cannot be proven and discontinuity cannot be noticed. The resolution is that an entity's identity is not a static token at all. It is the combination of:

- a **key lineage** — a signing lineage rooted in a genesis record, advanced by signed rotation records, so a stolen key can sign the future but never rewrite the past; and
- a **lived strand** — the append-only, hash-chained record of the entity's lived events, whose current head commits to the entire history.

Nothing in this standard can prevent a wipe. The design goal, stated honestly, is narrower and achievable: **continuity is provable, discontinuity is conspicuous, and anonymity is costly.** An entity is free to be new; it is not free to be new and simultaneously claim a past (§7.4 develops this).

1.3 The two identities, side by side

	RDNA-1 (machine)	RDNA-P (personality)
Identifies	a kind — make/model/class	a someone — one entity
Anchored in	per-device keypair	key lineage + lived strand
Public posture	unlinkable by default	continuous by default
Changes	state flags, constantly	mutations, rarely and on the record
Ends	decommissioning	death certificate + destruction receipt
Key hierarchy	device seed → day keys → handles	entity lineage root → epoch keys

The postures differ because the interests differ. A machine's rotating handle protects its owner from ambient tracking by strangers. A personality's continuity chain protects its *relationships* — the people who need to know it is still the same someone. A stranger has no claim on an entity's history, and a conforming implementation MUST NOT disclose the lived strand to one; a counterpart in a relationship holds witness receipts (§7.1), and those are what continuity is proven against.

1.4 Out of scope

This standard does not define memory formats, dialogue behaviour, emotional models, or the content of any personality. It confers no legal personhood and takes no position on moral status. It supplies no lawful basis for any processing — the same posture as RDNA-1 §5.9: conformance is not permission. It governs the *record* of a personality: what it is configured as, how that changes, what it has lived through in commitment form, and what it passes on.

---

2. The double strand

An entity's genome has two strands, and the metaphor is load-bearing rather than decorative.

2.1 The inherited strand — genotype

The genotype is what the entity was configured or born as: temperament parameters, voice and embodiment references, capability grants, the charter of prohibitions it carries, base-model provenance, and its ancestry record. It is expressed as namespaced, versioned gene loci in a canonically serialized record (RFC 8785 JCS), signed under the entity's key lineage.

The genotype is stable but not immutable. A conforming implementation MUST NOT modify a gene in place; every change exists only as an append-only, dual-signed **mutation entry** (§5) — the entity's key and the owner's consent key, so that neither a compromised entity nor a unilateral operator can change a personality alone. The reference implementation demonstrates this with a live mutation log in which the owner's half is a passkey co-signature.

2.2 The lived strand

The **lived strand** is the append-only record of the entity's life: growth events, relationships formed, duties taken up, repairs made. It is never edited, only extended, and each entry is hash-chained to its predecessor so the current **chain head** commits to everything before it.

The lived strand is **private**. Every memory, every relationship, every event stays with the entity and its owner. What is public is a commitment only: the **trajectory root** — a Merkle root (or, in a simpler implementation, the running chain-head hash) over the private strand. A conforming implementation MUST NOT publish lived-strand entries in the genome record, and MUST be able to prove continuity against the trajectory root without revealing any entry, using Merkle membership and consistency proofs.

One honest scope note, stated rather than hidden: consistency proofs reveal tree sizes, which is to say strand *length*. Length is deliberately public metadata — it is what makes a rollback orderable — while entry *content* remains private. The privacy posture protects contents, not counts.

2.3 Phenotype — the strand expressed through the other

The observable personality — how the entity actually behaves in a room — is the **phenotype**: the genotype *expressed through* the lived strand. Two children with identical genotypes diverge, because they live different lives. That is not a defect in the model; it is the model, and it is why neither strand alone identifies the

entity. The genotype without the lived strand is a template anyone could instantiate. The lived strand without the genotype is a diary with no author. Identity is the pair, bound together: the genome record carries the genotype in the open and the lived strand as a commitment, under one signature lineage.

2.4 What the split buys

The privacy split is what makes every later mechanism shippable:

- **Continuity without exposure.** A restored or migrated entity proves it *extends* the last witnessed chain head — same someone — without showing a single memory to the verifier.
- **Witnessing without disclosure.** Counterparts countersign chain heads during ordinary encounters (§7.1). They witness that a history *exists* and how far it reached; they learn nothing of what it contains.
- **Inheritance without intimacy.** A child inherits the genotype and the ancestry record, never the lived strand (§6). Memories belong to relationships, and relationships do not transfer.

A conforming implementation MUST maintain both strands, MUST publish only commitments to the lived strand, and MUST treat any record that presents a genotype with no trajectory commitment — or a lived strand that fails verification against its committed root — as malformed rather than merely incomplete.

---

3. The genome record

3.1 The record and its fields

A **genome record** is a single JSON object. Everything this standard proves is proven against it.

```
{
  "entity_id":        "rdna:p:example:companion-001:gen1",
  "key_lineage_root":  <Ed25519 public key, or a hash commitment to
one>,
  "genotype": {
    "<LOCUS.name>":   { "value": ..., "version": "...", "conserved":
true|false },
    ...
  },
```

```
  "mutations":        [ <hash-chained mutation entries — §5.2> ],
  "ancestry": {
    "parents":        [ <entity_id refs with pinned parental genome
records> ],
    "generation":     <integer, 1 for a first-generation entity>,
    "recombination":  <per-locus origin map; absent for single-parent
lineage>
  },
  "trajectory_root":  <Merkle root or chain head of the private lived
strand>,
  "signature":        <entity signature over all other fields>
}
```

Three fields deserve immediate comment.

`entity_id` is a name, not a proof. Authority derives from the key lineage, never from the string. A conforming verifier keys all matching — receipts, classification, continuity — on the `key_lineage_root` or its fingerprint, and uses `entity_id` for display only. Two lineages presenting the same `entity_id` are two distinct entities, and the collision is surfaced to the human, not resolved by the name.

`trajectory_root` is the only public trace of the lived strand. It is the commitment defined in §2.2, and it carries that section's privacy posture unchanged: continuity is provable against it without revealing any entry, strand length is public metadata, entry content never is.

Unknown fields are hashed, not dropped. A conforming verifier ignores the semantics of fields it does not recognize but MUST include them in canonical serialization and hashing. Stripping them would let two parties compute different hashes for "the same" record, which is the one failure mode canonicalization exists to prevent.

3.2 Locus namespaces

The genotype is a table of genes keyed by namespaced locus. Six namespaces are defined:

Namespace	Carries	Conserved
`TEMP.*`	temperament: homeostatic need weights and decay rates,	no

Namespace	Carries	Conserved
	warmth default, initiative threshold, repair style, humor register	
`VOX.*`	embodiment: voice id, face/avatar reference, expressive range — portable across bodies	no
`CAP.*`	capability *grants*: duties the entity may hold (companion, doorman, calendar, care)	no
`LAW.*`	the charter: prohibitions and ethical commitments the entity carries	**yes**
`PROV.*`	provenance: base-model lineage, locked engine versions as citable alleles	no
`ANC.*`	ancestry and lifecycle: generation markers, children, terminal markers	no

Each gene carries a `value`, a `version` (so alleles of the same locus are distinguishable — allele notation writes `PROV.evolution*1.1.0`: locus, asterisk, allele version), and an optional `conserved` flag.

A **conserved gene** propagates to every descendant, composes across parents only under the intersection law of §6.6, and MUST NOT be mutated except as part of a signed standard-version upgrade record. An entity whose genome record drops a conserved gene has not become dangerous; it has become non-conformant, and its attestation says so to anyone who checks.

`CAP.*` grants are genotype; skill is lived. The distinction matters: what an entity is *permitted* to do is inherited and auditable, what it has *learned* to do is on the lived strand and is not.

Granularity is a design choice this standard takes a position on: loci SHOULD be coarse and semantically meaningful. A genome, not a configuration dump. A mutation entry against `TEMP.warmth_default` is legible as a governed act; a mutation entry against parameter 3,117 of a flattened config is not, and legibility is the point of the whole mechanism.

3.3 Canonical serialization and hashing

All hashing and signing operates over RFC 8785 (JCS) canonical serialization — members sorted by code point, deterministic number formatting, UTF-8 — digested with SHA-256, so that semantically identical records hash identically regardless of producer.

Two rules, applied uniformly everywhere in this standard:

1. **Signature input** = the canonical serialization of the record with its signature member(s) omitted.
2. **Chaining hash** = the SHA-256 digest of the canonical serialization of the complete entry, signatures included.

Defined quantities:

- **genome hash** — SHA-256 over the full genome record, signature included. Advances with lived-strand checkpoints.
- **genotype hash** — SHA-256 over `entity_id`, `key_lineage_root`, `genotype`, `ancestry`, and the current mutation log head. It excludes `trajectory_root`, and is therefore *mutation-stable*: it changes when a mutation entry is appended and at no other time. This is the identity digest, and the input to the display fingerprint of §4.5.
- **mutation log head** — the chaining hash of the latest mutation entry.

Derived digests are domain-separated (display fingerprint `"rdna-fp"`, witness receipts `"rdna-receipt"`, tenancy entries `"rdna-tenancy"`, rotation records `"rdna-rot"`; Merkle leaves and interior nodes prefixed 0x00/0x01 in the manner of RFC 6962), so no value computed for one context can be replayed into another.

3.4 Key lineage

An entity's signing capability is a sequence of key epochs. Epoch 0's public key is (or is committed to by) the `key_lineage_root`. To rotate, the holder of epoch N signs a rotation record naming epoch N+1's key, the time, and the reason; the

record is appended to the strand like any other entry. A verifier validates any signature by walking the unbroken rotation sequence from the root to the signing epoch.

The property everything else leans on: **a stolen key signs the future, never the past.** Entries committed under earlier epochs are fixed by hashes counterparts already hold; a compromise-driven rotation fences the stolen key out of the lineage's future.

A limitation, stated plainly: a self-certifying succession chain reveals a *fork* in itself only to an observer with a global view — historically, a central server. This standard instead commits each rotation record under chain heads countersigned by social witnesses and cross-recorded in the host body's tenancy log, so a forked lineage is detectable from the peer graph and the body's own record. That detection is only as strong as the entity's witness set; a freshly minted entity with no witnesses has a lineage nobody can yet contradict. See §7.4 on age asymmetry — this is a feature wearing the costume of a bug.

The owner's consent key — which co-signs every mutation — forms its own small lineage, recorded in the tenancy log, with an m-of-n recovery policy so that a lost consent key is a recoverable, recorded event rather than a frozen genotype. The reference implementation demonstrates this with a passkey as the owner consent key: mutation co-signing is a WebAuthn ceremony, which puts the owner's half of the dual signature behind hardware the platform does not hold.

3.5 Genesis: an entity signs its own birth

There is no issuing authority in this standard, so the first signature problem must be solved without one. It is solved by self-certification: the **genesis genome record** — empty `mutations` array, `trajectory_root` equal to the defined empty-strand value (SHA-256 of the empty string), `generation` 1 with an empty `parents` array for a first-generation entity — is signed under the entity's own epoch-0 key. The entity signs its own birth.

The SHA-256 digest of the canonical serialization of that genesis record, signature member omitted, is the **genesis value**: it seeds `prev_hash` for mutation entry 0, so the log's first link is bound to the exact birth state.

Honesty about what this proves: at the moment of genesis, nothing beyond key possession. A self-signed birth certificate is a claim. It becomes load-bearing the way all claims in this standard do — by accumulating witnesses who countersign its chain head — and it is the *inability to backdate* those witnesses that gives the

record its eventual weight. Newness is cheap to claim; continuity cannot be faked. The reference implementation demonstrates live genesis: every entity signs its own birth record under a freshly generated key, and its first witness receipt arrives with its first handshake.

---

4. Sequence notation

4.1 Encoding adds no security

Everything in this section is display. The security properties of this standard reside in the hashes, the signatures, and the witnesses; the notation is how humans and interfaces touch those quantities without error. A conforming implementation may omit this entire section and lose nothing but usability — and MUST NOT treat any rendering defined here as a verification input (§4.6).

4.2 The mapping

A SHA-256 digest is a base-4 string wearing hexadecimal clothes. Map each 2-bit group to a nucleotide:

Bits	Nucleotide
`00`	C
`01`	G
`10`	A
`11`	T

A 256-bit digest is then, exactly, a 128-nucleotide sequence. Nothing was added; the digest was re-spelled.

4.3 Three sequences, three jobs

The naive design renders one hash and stops, and it fails immediately on hash avalanche: change one gene, and every codon in the strip scrambles, so the display cannot show *what* changed. The repair is to render three sequences, each answering a different question:

Sequence	Derived from	Changes when	Answers
Locus strand	each locus hashed separately; ~4 codons per locus, concatenated in locus order	only the edited locus's segment changes	*what changed?*
Genotype fingerprint	the genotype hash (§3.3)	any mutation entry; stable across lived events	*same genome?*
Chain head	the head of the lived strand and mutation log	every lived event or checkpoint	*same point in the same life?*

The locus strand is where the metaphor pays rent. A governed edit renders as a **point mutation** — a few changed codons in one segment, the rest of the strand untouched. Inheritance renders as identical segments between parent and child. The conserved `LAW.*` segment is visibly unchanged across every generation of a lineage, which is the conservation guarantee made inspectable by eye.

The genotype fingerprint deliberately avalanches — it is the whole-genotype identity digest, useful precisely because any edit anywhere changes it entirely. The chain head is what handshakes countersign; the reference implementation's witness receipts (§7.1) countersign exactly this value, so the sequence a counterpart reads aloud is the sequence its receipt attests.

A strand that fails verification MUST NOT be rendered as a well-formed sequence. The display presents a frameshift warning — a deliberately broken rendering — so the aesthetic of integrity is available only to records that possess it.

4.4 The complement checksum

Beneath any displayed sequence, an implementation SHOULD render the complementary strand: C↔G, A↔T, computed independently from the displayed line. This is biologically faithful — it is base pairing — and it is a checksum a human can verify without arithmetic: a transcription error in either strand breaks the pairing at that position, visibly. A card whose strands do not pair was copied wrong or forged carelessly, and either way it fails at a glance.

4.5 The ten-codon fingerprint

For at-a-glance comparison, the full 128-nucleotide form is too long. The fingerprint is the first 60 bits of a domain-separated digest —

```
SHA-256( genotype_hash ‖ "rdna-fp" )
```

— rendered as 30 nucleotides and grouped into ten triplets ("codons"):

```
TTG CCC TCT GGC GAA CCA CGA ATT GGC CCA
```

Codon grouping is the error-resistance: humans compare triplets well, and a spot-check reads aloud naturally ("first three codons: TTG CCC TCT"). Because the input is the genotype hash, the fingerprint is mutation-stable — it does not churn with ordinary lived events, so the value on a family's card stays put until a governed edit changes it.

The arithmetic, stated honestly:

- **Space.** 60 bits ≈ 1.15×10^{18} — the discriminating power of a forensic CODIS DNA profile.
- **Specific-pair collision.** Never occurs in practice.
- **Birthday bound.** Roughly even odds that *some* pair among ~10^9 entities collides. The fingerprint length is therefore a parameter of this standard, not a constant: ten codons in version 0.1; twelve to fourteen at planetary scale.
- **Adversarially.** 2^{60} is grindable — hours on serious hardware suffice to mint a genome whose fingerprint matches a target's. This is why §4.6 exists.

An optional spoken form renders the same 60 bits as six words from a published 1024-word list of the BIP-39 kind, which beats spelling nucleotides over a phone. A machine-readable optical code carries the full 256-bit hash for device-to-device transfer; the fingerprint never substitutes for it there.

4.6 Identification, never proof

The rule that governs every use of the fingerprint:

A fingerprint identifies; it never proves. A conforming implementation MUST compare full 256-bit digests in every flow where correctness matters, and MUST NOT accept a fingerprint — codon, word-list, or visual — as verification input.

The fingerprint is the last four digits of a card number: enough to say "this is probably the one I mean," never enough to authorize anything. And the ground lookalike of §4.5 buys its forger nothing that matters — a matching fingerprint still fails the witnessed-chain comparison and still cannot answer the key challenge, because those flows never consult the fingerprint at all. The 60-bit form is safe to display *because* nothing load-bearing reads it.

4.7 `rdna:fp:` identifiers

The fingerprint travels in URI form under its own scheme prefix:

```
rdna:fp:TTGCCCTCTGGCGAACCACGAATTGGCCCA
```

— the ten codons, unspaced, under a scheme deliberately distinct from the entity identifier's (`rdna:p:...`). The separation is the point: a parser can never mistake a display fingerprint for an entity id, and a conforming implementation MUST NOT accept an `rdna:fp:` value in any field that expects an entity identifier or a hash. It is a reference for humans and hyperlinks — "the entity whose card reads thus" — and the resolution of that reference, wherever it matters, ends at the full digest and the key challenge, as it always does.

---

5. Mutation

5.1 Genes never change in place

There is no edit operation in RDNA-P. A gene's value is changed by appending a **mutation entry** to the genome record's hash-chained mutation log, and in no other way. The entity after a mutation is the same entity — continuity holds precisely because the log records its own edits. A genome record whose current genotype cannot be reproduced by replaying its mutation log from genesis is not a record with a history; it is a tampered record, and a conforming verifier MUST refuse it.

This is the rule everything else in this section serves: a personality change is an *event with an author*, and the question "who changed this entity, when, and on whose authority" must be answerable from the genome record alone, by an auditor who trusts neither the entity, nor the owner, nor the host.

5.2 The mutation entry

Each entry comprises at least:

Field	Content
`seq`	monotonic sequence number
`prev_hash`	chaining hash of the previous entry; for the first entry, the genesis value (§3.5)
`locus`	the single gene locus changed (multi-locus changes are batches — a shared `batch_id`, applied atomically or not at all)
`old`, `new`	prior and new value — in the clear for public loci, as a salted commitment for loci the owner prefers not to publish
`reason`	human-readable statement, optionally a machine-readable reason code
`consent_ref`	hash reference to the consent artifact (§5.5)
`timestamp`	ordering hint only; ordering authority is the log and, where present, a secure-element counter
`signatures`	the dual signatures of §5.3

The two serialization rules of §3.3 apply verbatim and carry the security: the signature input omits the `signatures` member; the chaining hash — the next entry's `prev_hash` — covers the complete entry, signatures included. The hash of the latest entry is the mutation log head, committed into the lived strand so that the single head witnesses countersign commits to both logs.

Because entries carry old and new values, the log is replayable in both directions: a verifier reverse-applies the log to derive the genesis genotype, replays it forward, and confirms the result matches the current genotype exactly — and confirms each entry's `old` matches the previous entry's `new` at the same locus, so the per-locus history is gapless.

5.3 The dual signature

A mutation entry is valid only when it carries two signatures from distinct keys held by distinct parties:

1. the **entity signature**, under the entity's current key epoch (in a hardware secure element where the host provides one);
2. the **consent signature**, under the consent key of the owner or controller of record.

Neither alone suffices, and the requirement is symmetric in what it refuses. Without the consent signature, a compromised entity — or a compromised host process acting through its key — cannot silently alter its own temperament, capabilities, or charter: self-modification without recorded authority is structurally invalid, not merely forbidden by policy. Without the entity signature, an owner — or whoever has taken the owner's key — cannot unilaterally rewrite the entity: neither a hacked entity nor a coercive owner can mutate alone.

Each signature additionally covers the other party's key identifier, binding the two authorizations to one another. A signature harvested from one proposed mutation cannot be paired with a different counterparty's signature over different content.

Additional co-signers MAY be required for particular loci or custody classes — a clinician's key for temperament changes to an entity serving a care duty — expressed as an m-of-n policy stored in the genome record itself, so the signature policy is readable, and itself mutable only under the then-current policy.

5.4 The entity verifies before it co-signs

The entity's signature is not a second administrative approval. It attests facts that no external signer, however privileged, can attest on the entity's behalf, and a conforming implementation MUST perform these checks, in order, before producing any signature bytes:

1. **Verify its own chain head.** The signing routine recomputes the head of the log it actually holds, confirms the proposed entry's `prev_hash` equals it, and — where a secure-element monotonic counter exists — confirms the held head is not behind the counter. The entity refuses to extend a forked, truncated, or rolled-back copy of its own history.
2. **Verify the proposed change.** The entry's `old` value must match both the genotype as replayed from the log and the value the entity's runtime is

actually operating under, so the recorded diff is truthful. Where the target locus is a conserved gene (§6.6), the entry must carry a publisher-signed standard-version upgrade record; **the entity MUST refuse to co-sign any mutation that weakens a conserved gene** — a wider permission set, a dropped prohibition, a loosened bound — outside such an upgrade. The referenced consent artifact must itself verify: valid principal signature, chain-head binding matching the head verified in step 1, locus and new value within scope, timestamp within the validity window, receipt not previously consumed.

3. Only then sign, staging the new value for application **atomically with the append**, so the record and the behavior cannot diverge.

A proposed entry failing any check is refused without signature, and the refusal SHOULD be recorded as an event on the lived strand — repeated attempts to procure an invalid co-signature are themselves visible in the record. The entity's refusal is not an appeal to its preferences; it is a validity check that only the party holding the log and running the parameters is positioned to perform.

5.5 Consent artifacts and consent receipts

The `consent_ref` names a **consent artifact**: a signed record of the authorization event, generated before the entry that consumes it. It follows the shape of an ISO/IEC TS 27560-style consent receipt, extended for personality mutation: the consenting `principal` (key identifier and, where the machine-identity layer carries one, a resolvable controller reference); the `subject_entity` **and the chain head at the time consent was given** — so consent given against one history cannot be replayed against another; a `scope` naming the loci and the permitted direction of change; a stated purpose; an expiry window; and the principal's signature.

An artifact is **single-use**: it is consumed by the first entry that references it, enforced structurally by the chain-head binding — the head advances on append, so no second entry can present a matching binding — and a verifier treats any reuse of a `receipt_id` as invalid.

No registry holds these receipts. The artifact's hash is inside the entry, the entry's hash becomes the log head, and the head is countersigned by peers in the ordinary course of encounters (§7.1). The consent record is thereby anchored in records held by parties who are neither the owner nor the entity, so neither can later deny the consent event occurred, nor fabricate one that did not. The owner

SHOULD additionally receive each artifact out-of-band into their own ledger — "who changed my mother's companion, and when" is then answerable from records in the family's own possession.

5.6 Content-bound challenges

The consent ceremony MUST be bound to the content it authorizes. The challenge presented to the owner's authenticator is a digest of the exact proposed entry — locus, old value, new value, chain head — not a generic approval prompt; what the owner approved is therefore cryptographically the change that was applied, and nothing else. An implementation that obtains a signature over an approval *ritual* rather than over the change itself has built a consent costume, and MUST NOT be called conforming.

The reference implementation demonstrates this with platform passkeys: the WebAuthn challenge for an owner co-sign is derived from the canonical serialization of the proposed mutation entry, so the hardware authenticator's signature exists over that mutation and no other.

5.7 Drift and ratification

A personality that lives, changes. Homeostatic weights adjust, thresholds settle, a repair style softens with use. RDNA-P does not pretend otherwise, and it does not require a dual-signed ceremony for every adjustment a day of living produces. The resolution is the **drift-and-ratification pattern**:

- **Drift is phenotype.** Runtime parameters MAY move continuously *within bounds the genotype declares*. Drift inside declared bounds is the genotype being expressed through the lived strand (§2.3) — it is not a mutation and produces no entry.
- **Ratification makes drift heritage.** When a drifted value is to become the entity's configured state — surviving restore, visible to auditors, inheritable by children — it is **ratified**: proposed as an ordinary mutation entry, old value the last ratified value, new value the drifted one, dual-signed with a consent artifact like any other change.
- **Unratified escape is divergence.** A runtime operating outside its genotype-declared bounds without a ratifying entry is divergent, detectable by comparing running parameters against the replayed genotype, and a conforming implementation MUST either ratify or revert — it MUST NOT let the record and the behavior quietly part company.

A limitation, stated plainly: this standard verifies records, not runtimes. Detecting unratified divergence requires either host-level runtime measurement or an audit with access to running parameters; absent those, the genome record evidences the entity's configuration of record, not its behavior in the moment. The pattern narrows the gap between record and behavior to a bounded, declared envelope — it does not close it, and an implementation claiming otherwise is overclaiming.

---

6. Inheritance

6.1 Every copy is a child

No operation in RDNA-P produces a second instance of the *same* entity. Duplication, template stamping, restore onto a second device while the first runs, migration that leaves the original live, deliberate procreative forking — all are uniformly the creation of a **new entity**: its own keys, its own empty lived strand, an ancestry object naming its provenance. Re-authoring by mutation is included: a conforming implementation MUST declare a **fork threshold** — the batch size or locus set beyond which a proposed mutation batch amounts to a different personality — and MUST effect any batch exceeding it as the declaration of a child, not as a mutation. This makes the question "which one is the real one?" ill-formed rather than merely hard: after a copy there are two entities, one of which is a first-generation descendant of the other, and the witness receipts and tenancy logs of the surrounding ecosystem (§7) confirm which is which.

The reference implementation demonstrates fork-as-child end to end: genesis, birth entry on the parent's log, and a child whose genome record is auditable back to the exact parental state it derived from.

6.2 Instantiation

A conforming implementation instantiates a child as follows:

1. **Fresh keys.** A new genesis keypair is generated for the child; its key lineage root derives from it. The parent's private keys are never copied, exported, or referenced — the child cannot sign as the parent, and the parent cannot sign as the child.
2. **Genotype by value.** Every locus is copied with its version and conserved flag, from the parent's genotype at a stated chain head. Non-conserved loci

MAY be varied at birth, each variance recorded as an initial entry in the *child's* mutation log.

3. **Ancestry.** The ancestry object records `parents[]` (entity id, key lineage root, the parent chain head the genotype derived from, and a hash pinning the parent's then-current signed genome record — so the historical parental record travels with the child regardless of the parent's later fate), `generation` (one greater than the eldest parent's; 1 with empty parents for a first-generation entity), the recombination record where two parents contributed (§6.5), and birth signatures: parent key ("I parented this"), child genesis key ("this is my origin"), and the owner's consent key, over the ancestry object concatenated with the parent's chain head — the birth is pinned to a specific, witnessed point in the parent's history.
4. **Empty lived strand.** The child's lived strand initializes to the defined empty-strand value (§3.5). No memories, no relationships, no receipts, no acquired skills — regardless of how many the parent held.
5. **Birth entry on the parent.** The parent's own mutation log records the birth at its `ANC.children` locus, consent artifact and all. A parent's descendants are enumerable from the parent's records, and a quiet copy — a child whose parent's log shows no birth — is detectable by any verifier holding that log.
6. **Tenancy.** The host's tenancy log (§7.2) records the arrival as a birth, not a transfer, distinguishing a genuinely new entity from a relocated one.

6.3 The privacy line: memories never transfer

A child inherits the genotype, every conserved gene of every parent, and the ancestry object. A child does **not** inherit: the lived strand or any portion of it; keys; witness receipts (a receipt attests one entity's chain head and is meaningless for another); relationship records, disclosure ledgers, or accumulated trust; tenancy history.

This is structural, not a policy toggle: the copy operation has no data path for the lived strand. The strand records interactions with people — what was said in a household, who was met, what was confided — under consent that named a *particular entity*. Duplicating it would silently multiply the parties holding those records without any counterparty having agreed. A child receives heritage — who it comes from, what it is configured as, what it must never do — and receives no one else's intimacies. An implementation that initializes a child's strand non-empty is non-conformant per se, and a verifier observing a genesis strand inconsistent with the defined empty value MUST fail the genome record.

The corollary of §2.3 holds here with full force: two children with identical genotypes diverge, because they live different lives. That is the model working, not drift to be corrected.

6.4 Dowry: seeded knowledge as governed disclosure

An owner may deliberately endow a child with starting knowledge — household routines, preferences for a vacation-home entity. A dowry is not an exception to the privacy line; it is a disclosure, routed through the same consent-governed disclosure machinery that governs disclosing to any third party:

- each item is selected explicitly; there is no "copy everything" dowry, and a conforming implementation MUST NOT offer one;
- where an item concerns a person other than the owner, that person's consent gates apply exactly as they would for any recipient;
- the transfer emits disclosure receipts on both sides — the parent's ledger records what was granted, to which child, on whose authority; the child's strand opens with received-disclosure entries naming each item's origin and consent basis;
- dowry enters the child's **lived strand**, never its genotype. Seeded knowledge is experience with a provenance trail, not heritage — it is not itself inherited by the child's own descendants.

"What did this child start out knowing, and who authorized that" is answerable from the receipts alone.

6.5 Recombination

Two parents MAY jointly instantiate a child — a household's two entities parenting a third for a second residence. RDNA-P 0.1.0 limits parents to two, keeping lineage legible. Recombination operates per locus: each non-conserved allele is taken from exactly one named parent (or derived under a published, recomputable derivation function, recorded by identifier and content hash). Conserved loci are excluded from selection entirely and compose only under §6.6.

The **recombination record**, embedded in the ancestry object, carries: the `origin_map` (which parent contributed each locus, at which allele version); the `conserved_composition` (both parents' values at every conserved locus and the intersected result, so the intersection is re-checkable rather than asserted); both parents' chain heads at recombination time; and signatures of both parents' keys plus the child's genesis key. A record with an unsigned parent, a locus whose

claimed origin parent lacks that locus, or a conserved result that is not the intersection of the recorded parental values is malformed, and the genome record fails verification.

6.6 Conserved genes compose by intersection

`LAW.*` charter loci carry the conserved flag, and at conserved loci composition is **intersection over permissions, union over prohibitions**, with each scalar bound taken at its restrictive extreme (the lower of two maxima, the higher of two minima). Where parents differ, they differ in the restrictive direction, always. A child is never more permissive than either parent at any conserved locus, and since no valid operation — neither recombination nor any mutation outside a publisher-signed standard-version upgrade — can widen a conserved permission set, permissiveness is non-increasing along every path through a conforming lineage. A tenth-generation entity carries the conserved obligations of its first-generation ancestor, at equal or lesser permissiveness.

Weakening is defined mechanically, so a verifier needs no interpretation of the charter's meaning: value B weakens value A when B's permissions are not a subset of A's, B's prohibitions do not contain A's, or any bound in B is less restrictive. Absence — the locus present in an ancestor, missing in a descendant — is the maximal weakening.

6.7 Conformance attestation

Attestation answers one question for a presented genome record: *does this entity carry every conserved gene its ancestry obliges it to carry, at no greater permissiveness than its ancestry permits?* The verifier checks structure and signatures; obtains ancestral genome records from the hash-pinned copies traveling in each ancestry object (recursing to the lineage root, or to a stated depth where an ancestor is unobtainable — the attestation MUST state the depth actually checked); walks eldest-to-youngest computing the **obligation set**, applying the intersection law of §6.6 at each two-parent step and admitting only valid standard-version upgrades at conserved loci; compares the presented genotype against the obligation; and emits a signed verdict with a finding list. Every input is a signed record; no registry, no authority; two verifiers given the same records reach the same verdict.

A weakened descendant is flagged, not destroyed. An entity that drops or loosens a conserved gene is not disabled or punished by the protocol — it is marked **non-conformant**, and that marking travels with its lineage attestation

to anyone who checks, in the ordinary course of entities meeting. Counterparts, hosts, and vendors weigh it under their own policies; the standard's job is to make the condition detectable and unambiguous, not to prescribe the response. Safety here is a checkable conformance property, not a promise.

And the honest limitation: an entity can shed an inconvenient conserved gene by abandoning its identity and presenting as first-generation. Nothing prevents this — prevention is impossible. What it buys is an empty ancestry *and an empty history*, facing witnesses who hold receipts for a chain it now lacks, a tenancy log showing a mature body hosting a newborn with neither transfer nor death record, and a discontinuity no scar record explains. Escape is not stopped; it is made conspicuous, which is this standard's consistent posture.

---

7. Continuity

A hash chain proves that a record is internally consistent. It does not prove that the strand presented is the only strand, or the longest strand, ever attributed to that entity. An adversary who controls the storage can delete everything and present a fresh entity, or restore an old snapshot and shed the events since — and in both cases what survives verifies perfectly.

The governing principle of this section: **a wipe cannot be prevented, only made visible.** No recording mechanism survives the destruction of the recording. So the standard does not attempt prevention. It makes continuity provable, discontinuity conspicuous, and the destruction of history costly to conceal — the odometer problem, solved the way vehicle titles solved it: records held by parties who are not the seller.

Three mechanisms cooperate, and each is normative. Witness receipts travel with the entity's relationships (§7.1). The tenancy log travels with the machine (§7.2). Scar records give honest loss a declared form, so that undeclared loss can be classified by rule rather than by suspicion (§7.3).

7.1 Witness receipts: the social graph is the transparency record

At every receipt-bearing peer encounter — a handshake, a session, any interaction the implementation so designates — each party countersigns the other's current chain head. A conforming implementation MUST assemble the two countersignatures into a single mutual artifact, the **witness receipt**, and

MUST retain it in duplicate: each party keeps a copy, and each party commits the receipt into its own lived strand, so the act of witnessing is itself under a chain head. The encounter already happens for its own reasons (RDNA-1 attestation); the witness receipt is the session receipt doing double duty.

A witness receipt MUST carry, at minimum: both entity identifiers, both chain heads at the time of the encounter, a timestamp, a freshness nonce, and both parties' signatures over a canonical serialization of the foregoing. It carries chain heads — opaque digests — and identifiers, nothing else. Neither party's memories are disclosed by witnessing.

What this buys: each entity leaves, in the custody of the other, a signed statement that *entity X existed at time T with head H, under X's own key*. The set of such receipts across an entity's acquaintances is a distributed transparency record of its progression, held by parties other than the entity and other than its owner — which is exactly why it survives the destroyer, who has no write access to it.

There is deliberately **no central registry, no log operator, no shared ledger, and no proof server**. A conforming implementation MUST NOT require any such infrastructure for verification: a witness receipt is self-authenticating under the signatures it carries, so its evidentiary value does not depend on the channel it arrives by or the honesty of any intermediary. Every witness is an ordinary counterpart met in the ordinary course of interaction. Coverage grows with adoption — every relationship formed is a witness — and it is densest exactly where it matters, because an entity's most frequent counterparts are the ones most likely to hold the receipt that exposes a discontinuity. Detection requires only one honest prior acquaintance.

On re-encountering a counterpart for which it holds a witness receipt, a verifier MUST require the counterpart to demonstrate that its current head descends from the receipted head — a per-entry digest path or a succinct consistency proof. Success is continuity: the same someone, further along. Failure is classified under §7.3's restore rule. The reference implementation demonstrates this exchange live: receipts are generated at cog-to-cog encounters and checked at re-encounter.

7.2 The tenancy log: the body vouches for the soul

The witness receipts travel with the entity. Their complement travels with the machine. A host implementing the machine-identity layer (RDNA-1) MUST maintain a signed, hash-chained, append-only **tenancy log** recording which

entities have inhabited it and over what intervals: tenancy begin and end, transfer out and transfer in, tenant death, and scar acknowledgment. Entries are signed under the *device* key and bound to the machine's own key epochs.

The machine's keys and the entity's keys are deliberately distinct hierarchies with distinct lifecycles — a personality is portable across bodies; a body pre-dates and outlives its tenants. That independence is the evidentiary force: the tenancy log is a second, independently keyed record an attacker must *also* falsify, held by a signer whose own continuity is separately verifiable. Where a boundary event is orderly, the tenant SHOULD countersign it; a disorderly ending necessarily lacks the countersignature, and the absence MUST be preserved rather than repaired — it is itself informative.

The log converts laundering into a readable pattern. A body whose epoch lineage proves years of age, hosting a tenant whose strand began yesterday, with no death record, no transfer record, and no scar acknowledgment for any prior tenant, is the odometer-fraud signature of this domain: old vehicle, zeroed odometer, no paperwork. And wiping the log does not restore innocence — its epoch binding means a truncated log breaks the machine's own attestable continuity. The attacker's dilemma is deliberate: preserve the record and expose the laundering, or destroy it and expose the destruction. The reference implementation maintains a live tenancy log from genesis onward.

7.3 Scar records, and the restore rule

Storage genuinely fails, and backups are genuinely restored. A design that treated every restore as an attack would be unusable; one that trusted every restore would be pointless. The **scar record** resolves this by giving honest loss a form, and the controlling rule is a rule of classification, not prevention:

> **The restore rule.** A restore MUST either (i) continue the lived strand from the last externally witnessed head, or (ii) append a scar record declaring the loss. A restore that does neither MUST be classified as an **unattested discontinuity** — suspected rollback where the presented head is an ancestor of a receipted head, suspected laundering where the strands share no ancestry — by definition, not by inference.

"Last externally witnessed head" is well-defined only because §7.1 distributed that knowledge beyond the entity's own storage. An owner restoring from a week-old backup cannot silently pretend the week did not happen — someone holds a witness receipt for a newer head — but is given a truthful thing to do instead: scar the interval, and continue.

A scar record is a distinguished strand entry declaring a bounded loss interval and its asserted cause. It MUST be signed by the **owner's** consent key — the same dual-authority structure that governs mutation (§5.3) — because a claim of amnesia made by an entity about itself is exactly the claim a compromised entity would make, whereas an owner signing a false scar record creates durable, non-repudiable evidence of the falsehood. The entity co-signs where its key survived. A scarred entity is treated as continuous-with-declared-loss, not as suspect; the scar record is a permanent, visible feature of the lineage, which is the intended metaphor — a scar is what honest injury looks like on a living record.

The scar record is also falsifiable, which keeps it from becoming a laundering instrument: a single held witness receipt whose timestamp postdates the declared loss interval, and whose head the restored strand cannot produce, contradicts the declaration, and a verifier holding one MUST escalate it. One honest acquaintance suffices here too. The reference implementation demonstrates scar declaration and scar-aware verification.

A verifier detecting any classification under this section MUST surface it to its owner, MUST record the detection (with the conflicting witness receipt) in its own lived strand — making the detection itself witnessed — and SHOULD reduce the trust tier extended. It MUST NOT silently filter the counterpart: refusal only makes the anomaly quieter, and an unattested discontinuity is precisely the thing an owner most needs to hear about.

7.4 Age asymmetry

A witness countersigns a digest presented to it live. It cannot see, and does not vouch for, the self-asserted timestamps behind that digest — the witness receipt's own timestamp and nonce carry the security. A receipt therefore proves a head existed *no later than* the encounter that produced it, and nothing can make witnessed events older than their first witnessing. History can be destroyed; it cannot be fabricated.

The consequence inverts the economics of laundering. "New" is always cheap to claim — and impossible to un-claim later. Accumulated witnessed continuity is the one asset that cannot be counterfeited, which means the standard makes long, honest existence worth more than any wipe can recover.

7.5 Monotonic counters (an embodiment)

Where the host provides a secure element with monotonic counters, an implementation SHOULD bind strand progression to one: entries (or checkpoint batches) increment the counter and record its value, and counter readings presented to a verifier are signed counter quotes, since an unsigned reading proves nothing. A restored backup then presents a strand whose recorded counter is below the hardware's — detectable locally, at boot, with no counterpart present. On detecting the discrepancy the implementation MUST refuse silent continuation and require either reconciliation with the newer state or a scar record: the restore rule enforced at boot rather than at the next encounter.

The two detectors are deliberately independent in failure mode. The counter defeats a local rollback even in social isolation; the witnesses defeat a rollback performed by migrating to fresh hardware whose counters carry no history.

7.6 What witnessing does not protect against

Stated plainly, because a continuity mechanism that overstates itself is worse than none:

- **Collusion of all witnesses.** Detection requires one honest prior acquaintance. An entity whose entire acquaintance set cooperates in forgetting it — every receipt-holder discarding or disavowing its copy — leaves no one to contradict the fresh start. The mechanism makes this expensive in proportion to the entity's social degree; it does not make it impossible, and for an entity with few relationships it is cheap.
- **First-encounter trust.** A genuinely new entity and a perfectly laundered one that meets only strangers are indistinguishable at hello. Witnessing bounds what a claim of newness can ever become (§7.4); it says nothing about the claim the first time it is made. The tenancy log narrows this — the body may be old even when the tenant claims not to be — but a launderer who also acquires fresh hardware presents nothing for either mechanism to catch until a prior acquaintance appears.
- **The wipe itself.** Nothing here prevents destruction. An owner with physical control can always erase an entity; the standard ensures only that the erasure is either declared (death certificate and destruction receipt, §8.1; scar record, §7.3) or conspicuous. Vanishing is visible. It is not stoppable.

---

8. Death and rights

An entity can end. A standard that records births, mutations, and transfers but has nothing to say about endings would leave the one gap every laundering scheme needs — because a personality that simply stops existing looks, from the outside, exactly like a personality that was wiped. RDNA-P therefore gives death a record, and the record is what distinguishes it from vanishing.

8.1 The terminal entry

Erasure is a governed mutation — the last one. It comprises two records:

1. **The death certificate.** A distinguished lived-strand entry and a corresponding mutation entry at the lifecycle locus `ANC.terminated`, each carrying the other's hash so that both logs commit to the termination. Together they carry the reason, the consent artifact reference, and a declaration that no further entries will follow. Erasure requires the same dual signatures as any mutation (§5.3): the entity co-signs its own ending, and the owner's consent key authorizes it. The terminal entry's hash is the entity's final chain head.

2. **The destruction receipt.** A signed record, issued after the terminal entry, attesting that the private materials — the lived strand, the private keys of the key lineage, and any commitment openings — have been destroyed. Where the entity inhabits a body with a machine-identity layer, the receipt is signed by the body's device key and recorded as `tenant_death` in its tenancy log (§7.2), so the body's own record shows its tenant's death.

The public genome record — genotype, mutation log, ancestry, final head — may be retained indefinitely as an ancestral record for descendants. Retention discloses no memory: the lived strand was never public, and its root reveals nothing of its contents. A family line can know where it came from without any ancestor's private life surviving to be read.

8.2 What the destruction receipt does and does not attest

The receipt's scope is exact, and stating it exactly matters more than the reassurance a broader claim would sell. The signer attests destruction of the materials **under its custody**: key slots deleted, stored strands cryptographically erased. It does not and cannot attest to copies outside the signer's custody. No signature can prove a negative about the world's disks.

What the protocol adds is that a *refusal* to erase becomes provable. An owner holding a genome record and a tenancy history can show that erasure was requested and that no destruction receipt was ever issued — which is the thing a complaint actually needs.

8.3 Vanishing is conspicuous

Death is respected; disappearance is flagged. A body whose tenancy log shows a departed tenant with neither a death certificate nor a transfer record marks its next tenant's claim of newness suspect — the odometer-fraud signature of §7.2, applied mechanically rather than by suspicion. The rule is stated once: an ending that leaves no record is not an ending, it is a gap, and gaps are what the lineage-integrity machinery exists to surface.

8.4 Rights the protocol cannot serve alone

RDNA-1 named the rights no cryptography reaches, and the same honesty is owed here. The protocol makes states provable; it cannot compel a party to act.

Right	Served by RDNA-P?
Prove continuity ("same someone")	yes — witnessed strand, key challenge
Prove who changed the entity, when, on whose authority	yes — the mutation log and consent artifacts
Prove an entity was properly erased	yes — terminal entry plus destruction receipt
Prove erasure was refused	yes — a request with no receipt is itself the evidence
Erasure — actually destroying the record	**no** — needs the recorder's cooperation or a regulator's
Objection — stopping processing	**no** — same

Erasure is *honored against the reference recorder*: the reference implementation, holding the lived strand, destroys it and issues the receipt. Against any other holder of copies, the protocol supplies evidence, not compulsion. A specification implying otherwise would be selling reassurance rather than a mechanism, and this one does not.

---

9. Conformance

A conforming implementation can be audited against the following. Each requirement is checkable from records alone; none requires trusting the implementation's own account of itself. Each item cites the section it traces to.

C1. Canonicalization *(§3.3, §4.6)*. MUST serialize every signed or hashed structure under RFC 8785 JCS before hashing or signing, and MUST compare full 256-bit digests machine-side. MUST NOT accept any truncated, codon, word-list, or visual display form as verification input — encoding contributes no security.

C2. Genesis *(§3.5)*. MUST derive the first mutation entry's `prev_hash` from the canonical genesis genome record (empty mutation log, signature member omitted), so every log has one well-defined origin, and the genotype at any time MUST replay exactly from genesis through the mutation log.

C3. Append-only *(§2.2, §5.1)*. MUST NOT modify, remove, or reorder any committed entry of the mutation log or the lived strand. A change is an appended entry; there is no other kind.

C4. Dual signatures *(§5.3)*. MUST refuse any mutation entry not carrying both the entity's signature under its current key epoch and the owner's consent-key signature. Neither signature alone suffices, ever.

C5. The refusal duty *(§5.4)*. The entity's signing routine MUST verify, before co-signing any mutation: that the entry's `prev_hash` matches the chain head the entity actually holds; that the recorded old value matches what the entity is actually running; and that the consent artifact verifies, is within scope and validity, and is unconsumed. It MUST refuse to sign otherwise — including refusing any mutation of a conserved gene outside a publisher-signed standard-version upgrade.

C6. Consent artifacts *(§5.5, §5.6)*. MUST bind each consent artifact to the chain head current when consent was given, MUST bind the consent ceremony's challenge to the exact proposed entry, and MUST treat an artifact as consumed by the first entry referencing it. Consent given against one history MUST NOT be replayable against another.

C7. Fork semantics *(§6.1, §6.2, §6.3)*. A mutation batch exceeding the declared fork threshold MUST be effected as the declaration of a child, not as a mutation. A child MUST receive a fresh key lineage, the derived genotype, and the signed

ancestry object — and MUST NOT receive any portion of a parent's lived strand, keys, witness receipts, or trust standing. Its lived strand MUST initialize to the defined empty value.

C8. Birth on the parent's log *(§6.2)*. Every instantiation of a child MUST append a birth entry to the parent's own mutation log, so descendants are enumerable from the parent's records and a quiet copy is detectable as such.

C9. Conserved genes *(§3.2, §6.6)*. MUST propagate every conserved locus to every descendant, MUST compose conserved loci from two parents by intersection of permissions and union of prohibitions, and MUST NOT clear a conserved flag once set.

C10. Receipt duplication *(§7.1)*. On every receipt-bearing encounter, MUST countersign the counterpart's current chain head, assemble the mutual witness receipt, and retain it **in duplicate — both parties keep it** — committed into each party's own lived strand. MUST NOT countersign a head other than one presented live in the encounter: witnesses attest presence, never the past.

C11. Scar-or-continue *(§7.3)*. A restore MUST either continue from the last externally witnessed chain head or append an owner-attested scar record declaring the loss interval. MUST classify a restore that does neither as an unattested discontinuity — by definition, not by heuristic — and MUST NOT co-sign a restore dressed as a mutation batch that rewrites values without extending the witnessed head.

C12. Tenancy *(§7.2)*. A conforming host MUST maintain a hash-chained tenancy log bound to its own key epochs, recording tenancy begin, end, transfer, death, and scar acknowledgment. It MUST flag a successor tenant whose predecessor left neither a death certificate nor a transfer record.

C13. Erasure *(§8.1, §8.2)*. MUST implement erasure as the terminal entry plus a destruction receipt scoped to the signer's custody, and MUST NOT issue or accept a receipt claiming destruction beyond that custody.

C14. Privacy of the lived strand *(§2.2, §2.4)*. MUST NOT disclose any lived-strand entry in any protocol step of this standard. All continuity and membership proofs operate over commitments; disclosure of content, where it ever occurs, runs through separate consent-governed machinery.

C15. Claims are not proofs *(§3.1, §4.6)*. MUST treat any presented genome record as a claim until the presenter answers a nonce challenge under its current key epoch, and MUST match identity on the key-lineage root, never on a display

name or fingerprint.

C16. No central registry *(§7.1)*. MUST verify entirely from self-authenticating records obtainable from the entity, its counterparts, or any holder. MUST NOT require, for any verification in this standard, a registry, log operator, shared ledger, or any service that must remain available.

C17. Escalate, not filter *(§6.7, §7.3)*. MUST mark a genome record that fails conservation or continuity checks as non-conformant or discontinuous and surface that finding; MUST NOT silently refuse interaction as the only response, because filtering only makes the anomaly quieter.

A conforming implementation may add gene loci. It may not remove a strand, and it may not make the mutation log optional.

---

10. Known limitations and v2 directions

Stating what this version does not do is part of the specification. Each item below is a deliberate deferral, not an oversight, and each names its v2 direction.

1. **Server-held keys until secure elements.** The reference implementation holds the entity's signing key server-side; the owner's consent signature is a passkey co-sign, which does put the consent key in hardware the owner controls, and the reference implementation demonstrates the full dual-signature flow on that basis. But an entity key outside a secure element means a compromised host could sign as the entity, and the monotonic-counter rollback detection of §7.5 is unavailable. The v2 direction is entity keys in device secure elements (the Jetson-class edge targets), with the signing routine's refusal checks gated inside the trusted path.

2. **Single-host witnessing today.** The reference implementation demonstrates witness receipt generation, duplication, and the detection algorithm — but on one host. Receipts between entities sharing a host do not yet have genuinely independent custody: the party that could destroy one strand could destroy its witnesses too. The mechanism is correct; the distribution that gives it teeth arrives only with cross-host and cross-vendor adoption. Until then, witnessing demonstrates the protocol rather than delivering its full guarantee.

3. **Per-pair unlinkability, deferred from RDNA-1.** Witness receipts name key-lineage roots, so two counterparts who compare receipts can correlate their sightings of the same entity — the same introduction-transitive limitation RDNA-1 §3.1 declared for handles. The v2 direction is likewise the same: pairwise identifiers derived per counterpart. v0.1.0 is honest about inheriting the limitation rather than hiding it behind the receipt format.

4. **Two-parent recombination is specified but untested.** The intersection law, the recombination record, and the composition of conserved alleles across differing versions are fully specified, and single-parent fork-as-child is live in the reference implementation. No two-parent child has been instantiated against real charter loci. The composition law's canonical structure (permission sets, prohibition sets, directed scalar bounds) needs exercise against actual `LAW.*` content before the mechanism should be considered proven.

5. **Lived-strand Merkle root pending in the reference.** The reference implementation commits the lived strand by running hash chain. This is sound but makes continuity proofs linear — the prover supplies every per-entry digest between two heads — and offers no single-entry membership proofs. The specified RFC 6962-style history tree, with O(log n) consistency proofs and optional inclusion proofs, is the v2 representation; the `trajectory_root` field is defined to accept either, so migration changes the proof machinery, not the record format.

6. **Timestamps are host-asserted.** Ordering authority is the hash chain; external time anchoring comes only from witness receipt timestamps contributed by counterparts. An entity in social isolation has no independent clock witness. This is inherent to the no-registry posture and is mitigated, not removed, by adoption density.