The Anchor Protocol:

A Mechanically Analyzed Control Plane for Local Agent Swarms

Erich Owens

Curiositech LLC

engineering@portdaddy.dev

September 2026
Version 1.5 (textbook edition)

Abstract

Agents sharing one machine cannot be trusted to behave, so the Anchor Protocol makes every agent prove cryptographically which capabilities it holds and for how long — a claim proved symbolically about the protocol, and checked mechanically against the Rust code that actually runs. Express lane: that sentence is the claim; the model-checked properties 3.1 and D.1 are the formal statements, and §5.3 is where they stop. A reader in a hurry needs nothing else.

As AI agents transition from isolated copilots to collaborative, autonomous swarms, local development environments face a crisis of coordination and trust. Existing tools designed for human-driven, static infrastructures lack the primitives required to prevent dynamic agents from corrupting shared state or squatting on ephemeral ports.

This paper introduces the **Anchor Protocol**, a cryptographic and semantic identity framework built into the Port Daddy daemon. We detail the protocol's evolution from symmetric MACs to Ed25519 signatures and multi-hop delegation chains inspired by Macaroons [4]. ProVerif [1] checks symbolic authentication and attenuation properties of the protocol models; Kani checks that the Rust core's parsing and comparison paths cannot panic on bounded symbolic inputs, with the cryptography stubbed; runtime conformance tests cover the deployed bridge. These layers do not amount to a proof of the entire implementation or of hardware-level constant time, and the multi-hop proof is structurally bounded rather than session-unbounded — Section 5.3 records those boundaries explicitly.

Keywords: distributed systems security, formal verification, capability-based security, multi-agent coordination, symbolic protocol analysis, ProVerif, JWT security, Ed25519

Reading time: approximately 30 minutes end-to-end. The Bonded Commons: Pre-Transactional Trust Infrastructure for Multi-Agent Systems *(Owens 2026) carries the economic and governance argument; either paper can be read first.*

Port Daddy Coordination Papers **Chapter 2 of 8**

Part I Ground Truth	1. The Single-Writer Kernel · **2. The Anchor Protocol** (this chapter) · 3. The Sealed Harbor
Part II The Cost of Seeing	4. The Legible Swarm
Part III What Survives the Restart	5. From Spawn to Person
Part IV Trade Between Strangers	6. The Harbor Economy · 7. The Bonded Commons · 8. The Federated Harbor

The Harbor, the Person, and the Economy reads in this order: each chapter stands on the ones before it, and each proof chapter follows the chapter whose promises it keeps. Every chapter is independently readable.

Contents

1 What problem this solves

The scene. It is 2 a.m. and your laptop is running four coding-agent sessions at once: one refactoring the auth module, one running the test suite against a local Postgres, one building a preview deployment, and one you started an hour ago and forgot about. All four are agents. All four can read your files, bind ports, and run shell commands with your full user permissions — and none of them can tell, from the outside, which of the other three is behaving. Nothing on the machine distinguishes the forgotten session from a process that has quietly replaced it.

That is the local swarm problem. In a multi-agent development environment, agents operate concurrently to read files, spin up test servers, and execute commands, and this introduces three concrete failure modes on `localhost`:

1. **Port Squatting (The "Ghost in the Harbor"):** If Agent A crashes, a malicious or malfunctioning Agent B can bind to Agent A's previously assigned port, intercepting traffic meant for A.
2. **Resource Contention:** Agents fighting over shared resources (e.g., a SQLite database file) without a centralized locking mechanism cause state corruption.
3. **Privilege Escalation:** A "Reviewer" agent delegated by a "Coder" agent should not possess the rights to initiate production deployments, yet local environments rarely enforce principle-of-least-privilege between processes.

The idea in one breath. *Every agent on the machine carries a short-lived, cryptographically signed card that names exactly what it may do; anything it hands to a child is strictly smaller; and any card can be withdrawn before it expires.* Everything that follows is that sentence made precise, then mechanically checked.

To make it enforceable, Port Daddy introduces the concept of a **Harbor**—a zero-trust, namespace-isolated execution environment where agents must prove their identity and capabilities. Our approach builds upon foundational work in capability-based security by Dennis and Van Horn [7] and the principle of least privilege articulated by Saltzer and Schroeder [15]. This protocol sits in a specific lineage — Macaroons for delegation, ProVerif and Tamarin for symbolic verification, the JWT algorithm-confusion literature for the Phase 1 hardening, and the emerging agentic-commerce credential profiles for the envelope — traced in full in Appendix F, which a reader following the argument rather than the citations can skip entirely.

How to read this chapter. §2 is the protocol: four phases, each closing a hole the previous one left open. §3 is the evidence: three verification layers, what each one proves, and what it hands the next. §5 is the accounting: the adversary we assume, the properties that survive it, and where the guarantees stop. Table 2 is the map for §3; Table 4 closes the loop on the three failure modes just listed.

Volume Context. This chapter provides the cryptographic substrate. *The Bonded Commons* builds the economic and game-theoretic layer on top: bonded participation, conditional auctions, mechanism analysis, and Sen-inspired advisory conflict resolution. The Anchor Protocol is a self-contained subset consumed as the trust kernel for the wider mechanism design. Read this chapter for "how does the daemon authenticate an agent?"; read *The Bonded Commons* for "why should there be a daemon at all?"

2 The Anchor Protocol Architecture

The Anchor Protocol defines how agents authenticate to a Harbor and to each other. It issues **Harbor Cards** (highly constrained JSON Web Tokens) and evolves them across four phases, each adding a capability that closes a specific attack surface from the previous phase (Table 1).

Table 1: Threat closure is cumulative, not substitutive. Each phase closes one attack surface, and the last column is why no later phase reopens it: every phase adds a check that a later phase changes the key or the chain of, never the party that performs it. [internal]

Phase	Mechanism	Attack surface closed	Stays closed because
1	HS256 pinning: the verifier takes the algorithm from policy	header-selected algorithm (algorithm confusion)	the algorithm is never read from the token; later phases change the key type, not who chooses
2	Ed25519 identity: the daemon signs with a private key, verifiers hold only the public key	shared-secret theft	no verifier holds a secret that could mint a card; a stolen public key mints nothing
3	delegation: a child card carries its parent chain, checked as a subset at every hop	downstream privilege growth	the subset check is per hop and monotone, so no later hop widens scope or lifetime
4	revocation: epoch filters gossiped between daemons and consulted at every acceptance	post-issue authority (a leaked card that should die before its TTL)	acceptance requires the current epoch, so every daemon that holds the epoch refuses the revoked card

A common thread across all phases is *capability attenuation*: a delegated token never carries more authority than its parent, and TTL only shrinks. Temporal attenuation is the same rule applied to the clock: a child's expiry is the earlier of its parent's expiry and the lifetime requested, so a twenty-minute child requested under a parent with ten minutes left is refused, and the acceptable narrower card is the one that inherits the parent's ten; in the capability lattice, lifetime is one more coordinate that moves down or stays. Figure 1 shows the nested-cap structure that the verifier enforces on every chain.

Numbers by hand — a child card under a parent with ten minutes left. A parent card holds `fs:read` and `db:write` with 10 minutes of lifetime remaining. A child requested with `db:write` for 20 minutes is refused: its rights are a subset, but its expiry would be later than the parent's. The same child requested for 8 minutes is issued with an 8-minute expiry; a child requested for 20 minutes *under the parent's clock* is issued with the parent's 10. In the lattice both coordinates moved down or stayed: rights $\{\texttt{db:write}\} \subseteq \{\texttt{fs:read}, \texttt{db:write}\}$ and lifetime $\min(20, 10) = 10$. Figure 1 draws the same check with the numbers 60, 15 and 5. [internal]

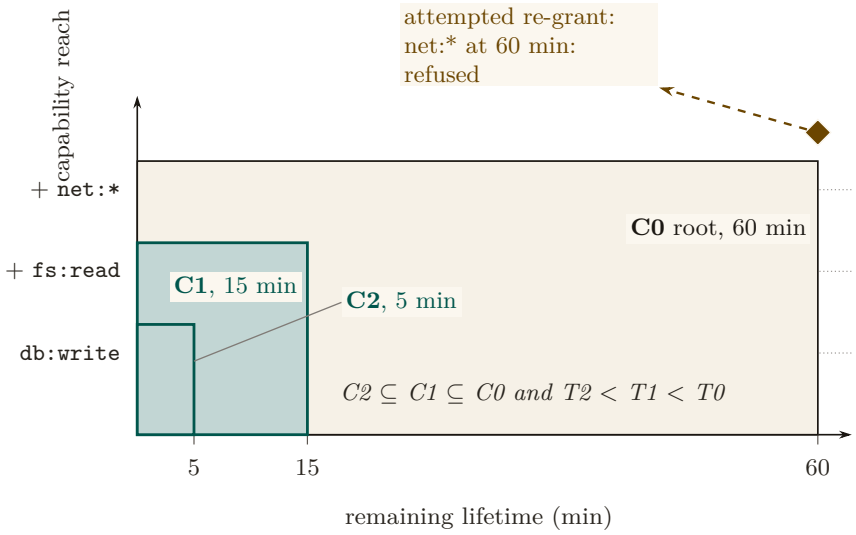


Figure 1: Nested rectangles for the capability envelope C0 (root, 60 min), C1 (15 min) and C2 (5 min): each child rectangle sits strictly inside its parent on both axes. The amber datum is an attempted re-grant of `net:*` at the full 60-minute root lifetime, outside C0 on the capability axis; the verifier refuses it because attenuation is conjunctive on rights and time together, not a trade between them. [internal]

2.1 Phase 1: Symmetric Pinning

Early versions of the protocol relied on HS256 (HMAC-SHA256). The primary vulnerability in JWTs is "Algorithm Confusion" (CVE-2015-9235 [11], CVE-2023-48238 [13]), where an attacker modifies the token header to `{"alg": "none"}` or symmetric HS256 while using an asymmetric public key as the HMAC secret.

Design invariant 2.1 Algorithmic pinning. The Port Daddy Verifier employs strict **Algorithmic Pinning**. It entirely ignores the user-provided `alg` header and explicitly forces the underlying crypto library to evaluate the signature using the pre-determined algorithm. This approach is recommended by the JWT Best Current Practices draft [14].

Figure 2 shows the two verifier paths side by side: the naive verifier trusts the `alg` field on the wire (and is exploited), while the pinned verifier records the issuance algorithm out-of-band and admits no rewrite trace.

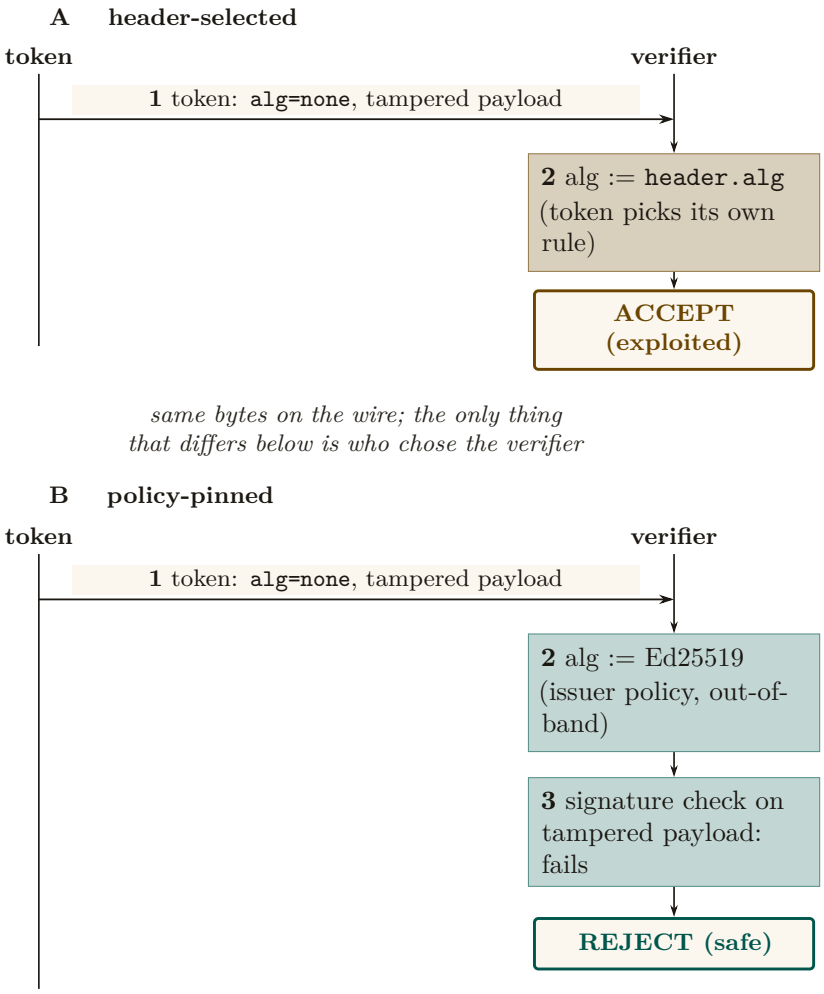


Figure 2: Two verifier ladders for the same wire bytes: **token** sends `alg=none` with a tampered payload to **verifier** in both panels. Panel A lets the token's own header choose the algorithm and message 2 accepts the forgery; panel B pins the algorithm to Ed25519 from issuer policy, so message 3's signature check fails and the token is rejected. This is the design invariant behind the ProVerif v1 (HS256) query $\neg$ attacker(*master_key*) holding TRUE: the accepting path in panel A is exactly the path Algorithmic Pinning removes [verified, v1_refined.pv].

2.2 Phase 2: Asymmetric Identity (Ed25519)

To support decentralized Agent-to-Agent (A2A) communication without sharing HMAC secrets, the protocol transitioned to **Ed25519 (EdDSA)** [8, 9]. Ed25519 provides fast, deterministic signatures with small keys and signatures, making it ideal for local agent communication.

Definition 2.1 (Harbor Key Hierarchy)**.** The Port Daddy Daemon acts as the Root Certificate Authority (CA). Each Harbor is assigned a unique Ed25519 keypair. The Daemon issues Harbor Cards signed by the Harbor's private key. Agents use the Harbor's public key (broadcast via Port Daddy's ambient discovery service) to verify tokens presented by peers.

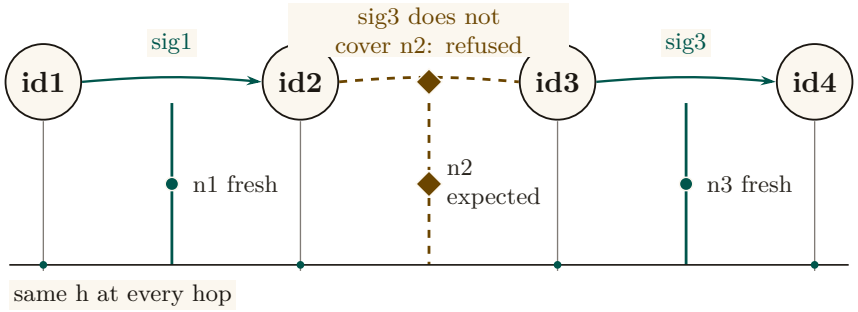
2.3 Phase 3: Stigmergic Delegation (The "Biscuit" Pattern)

Port Daddy v4 introduces multi-hop delegation. When Agent A spawns Agent B, it does not need to request a new token from the Daemon. Instead, it uses **Offline Attenuation**, inspired by Macaroons [4].

Definition 2.2 (Offline Attenuation)**.** Agent A takes its existing Harbor Card, appends a new set of restricted capabilities (e.g., `["db:read"]` reduced from `["db:read", "db:write"]`), and signs the *entire chain* with its own ephemeral private key.

The Harbor verifies the Daemon's root signature, Agent A's delegation signature, and mathematically ensures that Agent B's capabilities are a strict subset of Agent A's.

For depth-N chains, the verifier walks the structure shown in Figure 3. Each hop must produce a fresh nonce and bind the previous and next identities into its signature so that an attacker who intercepts a single hop cannot splice it into a different chain.



each ordinary crossing binds parent, child, a fresh nonce, and the same task hash h; the amber crossing reuses sig3 (bound to n3) where n2 was expected, so verification fails there

Figure 3: The delegation chain as a signed provenance braid across four identities id1–id4. Ordinary hops 1 and 3 bind parent, child, a fresh nonce (n1, n3), and the shared task hash h below. Hop 2 shows a splice attack: an attacker moves the legitimate signature sig3 (computed over id3, id4, and n3) into the id2–id3 position, where the verifier expects a signature over n2; the check fails at exactly that crossing, so the discontinuity is visible at one adjacent-identity boundary rather than anywhere else in the chain. [internal]

2.4 Withdrawing a card before it expires

Capability tokens have a TTL, but natural expiry is not always sufficient. Two circumstances force early withdrawal: (i) *compromise*—a Harbor Card leaks from its agent's process, and every second until TTL is a second the attacker has legitimate capabilities; and (ii) *policy reversal*—a principal decides an agent should no longer hold `db:write`, and waiting for TTL is unacceptable. A naive revocation list is $O(n)$ per verification and grows monotonically. While the Anchor Protocol previously attempted to gossip cuckoo filters directly between daemons — merging two peers' filters by OR-ing their bit arrays — it now gossips an **Append-Only Event Log of Revocation IDs** using Vector Clocks. To maintain $O(1)$ read performance, each daemon continually projects its event log into a local, ephemeral cuckoo filter [25] (Figure 4).

Why filters cannot be merged bitwise. The abandoned design failed for a structural reason worth stating plainly, because it is the reason the log exists. Two cuckoo filters cannot be safely combined by OR-ing their bit arrays. A fingerprint's final resting bucket is not a function of the key alone: it is the first candidate bucket i_1 if that bucket had a free slot at insertion time, and otherwise the alternate i_2 reached by evicting some other fingerprint — so occupancy depends on insertion order and eviction history, not just on the set of keys. Peer A may hold fingerprint `c1` in bucket i_0 while peer B, having inserted a different key first, holds the same `c1` in bucket i_3. The bitwise union contains both placements, which no single consistent filter state could have produced: subsequent deletions then remove the wrong occupancy and can *reanimate* a revoked card — a false negative, the one error a revocation gate must never make. Gossiping the append-only log of `kid`s and rebuilding each daemon's filter locally sidesteps the problem

entirely, because set union over identifiers is well defined where union over their lossy encodings is not.

Cuckoo filters in one paragraph. A cuckoo filter is a compact probabilistic set: under successful insertion and correct deletion discipline it has false positives but no false negatives, while supporting deletion. Each inserted key is reduced to a fingerprint and placed in one of two candidate buckets; on collision, fingerprints are relocated until a slot opens or insertion fails. Lookups inspect both buckets. The expected lookup cost is $O(1)$ and the approximate false-positive probability for two buckets of size B and f fingerprint bits is at most about $2B/2^f$ in the low-collision model. High load increases insertion failures; the implementation rejects or rebuilds rather than silently dropping a revocation. Duplicate fingerprints require counted or authoritative-table-backed deletion so that removing one key does not reanimate another.

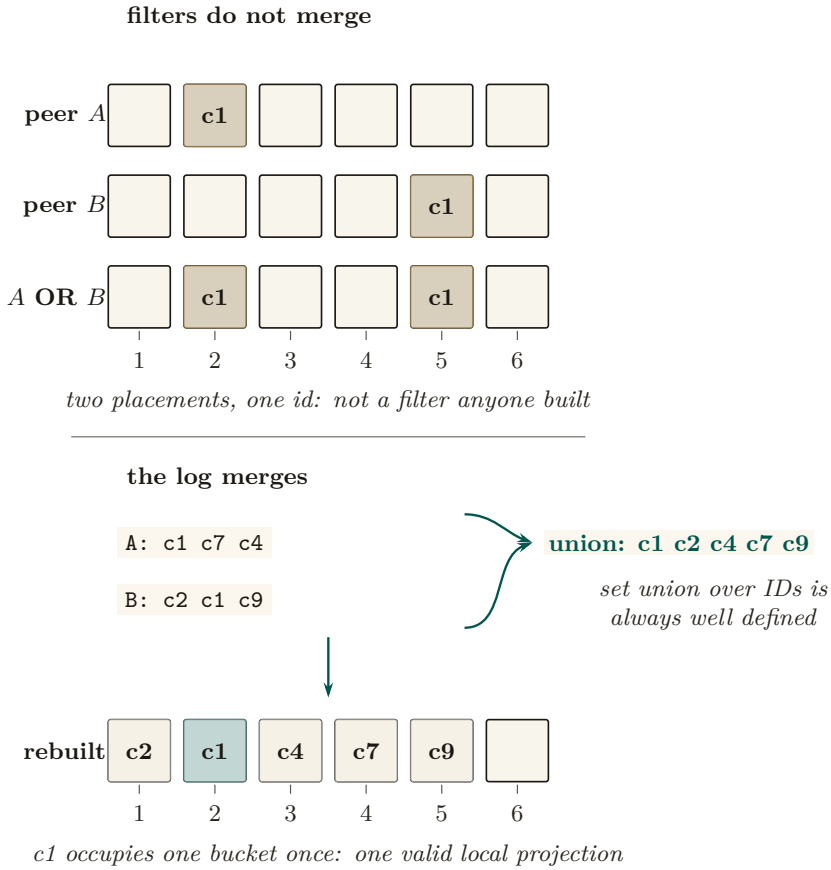


Figure 4: Two six-bucket cuckoo filters compared before and after the design change. Top: peer A inserts `c1` into bucket 2 and peer B inserts the same `c1` into bucket 5, an artifact of each peer's own insertion and eviction history; bitwise `OR` keeps both placements, which is a state no single filter build could have produced. Bottom: gossiping the append-only ID log instead—peer A's `c1 c7 c4` and peer B's `c2 c1 c9`—and taking the ordinary set union gives `c1 c2 c4 c7 c9`; each daemon rebuilds one local filter with `c1` in a single bucket. [internal]

Why cuckoo, not Bloom. For uniform-TTL workloads, a rotating Bloom filter would suffice. Port Daddy's TTLs are heterogeneous—Shipwright proposals run hours to days, `pd spawn` children run minutes, one-shot agents run seconds. Rotating Bloom either rotates at the longest TTL (short-TTL revocations linger orders of magnitude longer than needed) or buckets by TTL class (reintroducing the structural complexity cuckoo collapses). Cuckoo provides $O(1)$ per-entry removal at each specific expiry, plus the same removal handles operator-initiated early-revocation undo cleanly.

Space. At false-positive rate p_{fp}, a cuckoo filter uses approximately $\log_2(1/p_{\text{fp}}) + 3$ bits per entry. For $p_{\text{fp}} = 10^{-3}$, ≈ 13 bits per revoked card. A fleet retaining 10^5 active revocations fits in $\approx 160\,\text{KB}$—trivially in memory, trivially gossiped.

Numbers by hand — sizing the revocation filter. At $p_{\text{fp}} = 10^{-3}$ the per-entry cost is $\log_2 1000 + 3 = 9.97 + 3 \approx 13$ bits. A fleet holding 10^5 live revocations therefore carries 1.3×10^6 bits $= 162{,}500$ bytes, about 160 KB, and a gossip round that ships the whole filter rather than a delta moves less than a small image. At $p_{\text{fp}} = 10^{-6}$ the same fleet needs $\log_2 10^6 + 3 \approx 23$ bits per entry, $\approx 288\,\text{KB}$: three orders of magnitude fewer false positives for less than twice the space, which is why the filter's bound, not its size, is what the load ceiling protects. [internal]

Definition 2.3 (Revocation-Augmented Verification)**.** Let R_t denote the authoritative Append-Only Event Log of revoked Harbor Card IDs at time t. Each daemon d maintains a local, ephemeral cuckoo filter index $F_d \approx R_t$. Every Harbor Card verification additionally:

1. Looks up the card's `kid` in F_d. If absent, admit.
2. If present, query the local `revocations` table (authoritative). If genuinely revoked, reject with `revoked`; otherwise the filter is in the false-positive regime and the caller is admitted, with an asynchronous reconciliation job repopulating the filter.

Gossip-based synchronization. Daemons in a mesh synchronize revocation deltas by anti-entropy gossip [26]. Under a connected complete-overlay model with independent uniform peer selection and reliable rounds, epidemic dissemination is expected $\Theta(\log m)$ rounds. This is an expectation, not a deadline; partitions, message loss, topology, and scheduling change the constant or prevent global convergence until the partition heals. Security-critical revocations can additionally trigger best-effort immediate push. Figure 5 illustrates propagation across five daemons; it is a trace, not a timing proof.

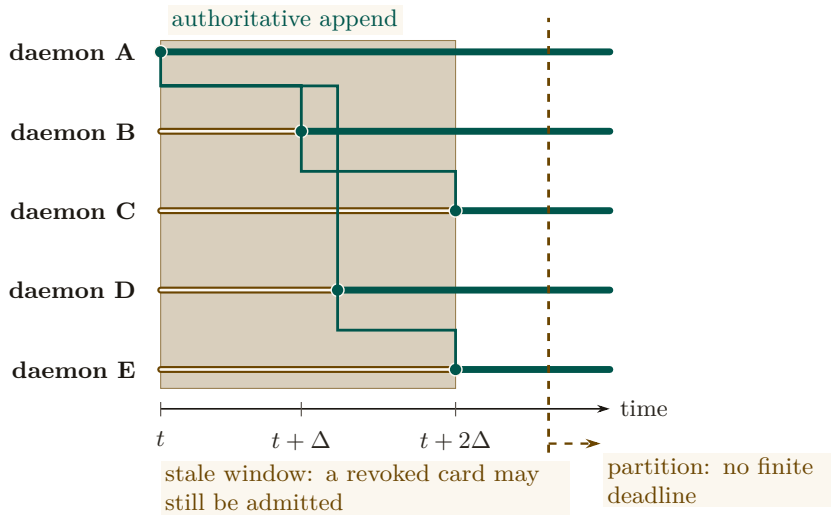


Figure 5: A revocation advances as an observation frontier, leaving a distinct stale-exposure interval on every verifier lane. This execution converges after two rounds; the protocol claim is only expected $\Theta(\log m)$ dissemination under connected reliable rounds. The dashed partition marker separates that expectation from a guarantee: an unbounded partition removes every finite delivery deadline.

Pollution as denial of revocation. The load ceiling keeps the false-positive bound honest; it does not by itself defeat an attacker who submits many authenticated-looking revocation identifiers to push the filter out of its bound-holding regime, which turns the revocation path into a denial of service. The design requires three things of the daemon: a per-issuer insertion quota, an overflow record for every refused insertion, and lookups that consult the authoritative

event log while a flooded filter rebuilds, so that flooding degrades revocation checks to slower, never to missed.

Design invariant 2.2 Filter monotonicity over a TTL epoch. For any Harbor Card c with issue time t_0 and TTL τ, if $c \in R_t$ for some $t \in [t_0, t_0 + \tau]$, then $c \in R_{t'}$ for all $t' \in [t, t_0 + \tau]$. After $t_0 + \tau$, c may be removed from R since the card has expired.

Design invariant 2.3 No partial reanimation. A revoked card cannot be un-revoked within its TTL. If an operator decides the revocation was erroneous, they must issue a new card.

Theorem 2.4 Conditional gossip expectation. Under the complete-overlay, uniform-peer, reliable-round model above, a revocation disseminates to all m daemons in expected $\Theta(\Delta \log m)$ time. No finite worst-case freshness bound holds under an unbounded partition.

Numbers by hand — how many rounds until every daemon knows. Five daemons, one revocation, a round time Δ of two seconds. Epidemic dissemination reaches all m peers in expected $\Theta(\log m)$ rounds; with $\log_2 5 = 2.32$ the expectation is between two and three rounds, so between four and six seconds, which is what Figure 5 draws as the interval from t to $t + 2\Delta$. Double the fleet to ten daemons and $\log_2 10 = 3.32$ adds one round, not five. Cut one daemon off from the others and no number of rounds reaches it: the expectation is conditional on the overlay staying connected, which is the sentence the theorem ends with. [internal]

Soundness preservation. Revocation strictly narrows acceptance, so existing ProVerif soundness proofs (Property 3.1) are unaffected. The new proof obligation is small: revocation is *enforced*, which follows by inspection of the verification path—the revocation check is unconditional and precedes admission. The card lifecycle gains a state (*valid* → *revoked* → *expired*) modeled as an additional transition (Figure 6).

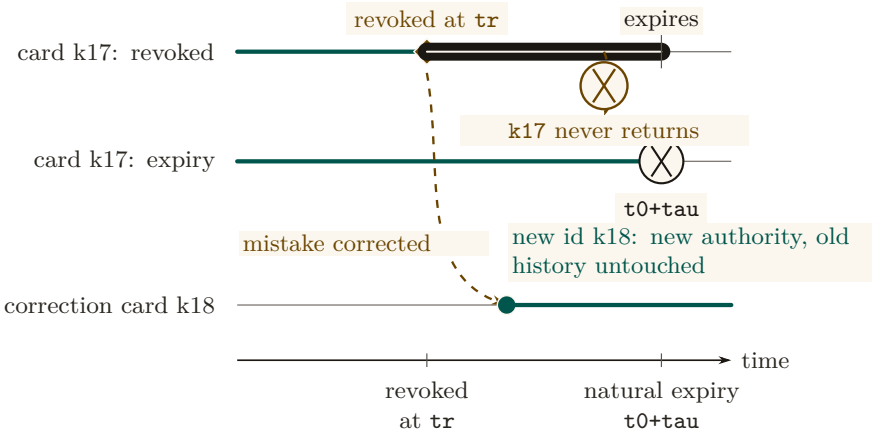


Figure 6: Acceptance is a one-way interval attached to one card ID. Card k17 stops being admissible either at explicit revocation (tick `tr`) or at natural expiry (tick `t0+tau`) and never re-enters the valid set. If revocation was a mistake, the operator issues k18 on a new lane: new authority on a new ID, with the rejected history on k17 left untouched rather than mutated. [internal]

Privacy. A Dolev-Yao adversary observing all gossip traffic learns the revoked-card IDs, leaking which agents have been disciplined. For an organization's own daemons, this is acceptable audit transparency. Stronger privacy is available by encoding revocations as salted hashes $H(\texttt{kid}, \texttt{salt})$ where the salt rotates daily, stored encrypted under a harbor session key.

Recall.

1. Without looking: state the one property that must hold between every delegated Harbor Card and its parent, across all four phases.
2. Why can two daemons not merge their cuckoo filters by OR-ing the bit arrays, and what do they gossip instead?
3. Name the two circumstances that force a card's withdrawal before its TTL naturally expires.

3 How we know it is correct

Following the industry standard set by AWS (s2n-tls [20]) and Microsoft (Project Everest [22]), we decoupled the verification of the *protocol design* from the verification of the *implementation*. The full stack has three layers (Table 2): symbolic protocol analysis at the top, bounded model checking on the Rust source in the middle, and runtime conformance tests against the deployed binary at the bottom. Each layer's obligations flow downward; each layer's evidence flows back up. The stack is sound only if every bridge — symbolic-to-source and source-to-deployed — holds.

Table 2: The assurance case, layer by layer. Obligations descend the rows: each layer hands the next a claim it cannot discharge itself. Evidence ascends them: each tool's output narrows what the layer above must assume. The last column names the two gaps that no tool closes; only a human asserts that the modeled protocol is this source, and that the inspected source is this running binary. [internal]

Layer	Tool	What it shows	What it cannot show
protocol model	ProVerif	secrecy, authentication correspondence in the modeled phase, attenuation soundness on the multi-hop model	that the modeled protocol is this source (the specification gap; a human asserts it)
Rust source	Kani and tests	the subset logic on two concrete vectors, bounded absence of panic with the cryptography stubbed, branch-freedom at source level	that the inspected source is this running binary (the deployment gap; a human asserts it)
running binary	conformance suite	the selected attacks are refused at the FFI boundary	any attack the suite does not contain

Table 2 is this section's map. Read down the rows for what each layer hands the next as an obligation it cannot discharge itself; read the third column for what has actually been checked about the running code. The last column holds the only two places in the whole assurance case where a human, not a tool, asserts that one layer's claim carries over to the next — which is exactly why they are listed as assumption gaps rather than as inferences.

3.1 Symbolic Analysis with ProVerif

To prove the protocol's logical soundness against a Dolev-Yao adversary [6] (an attacker who controls the entire network), we modeled the Anchor Protocol in **ProVerif** [1]. ProVerif represents protocols as Horn clauses and uses resolution to prove security properties for an unbounded number of sessions.

Model-checked property 3.1 Authentication correspondence in the modeled phase. For the modeled delegation phase, ProVerif confirmed the correspondence:

$$\begin{aligned}\text{Accepted}(b, h, cap) \implies\ & (\text{IssuedRoot}(a, h, cap) \wedge \text{Delegated}(a, b, cap)) \\ \vee\ & \text{IssuedRoot}(b, h, cap).\end{aligned} \tag{1}$$

The displayed query is a non-injective authentication correspondence: every accepted event has a matching modeled issuance/delegation event. It does not by itself prove replay freedom, capability attenuation, or arbitrary-depth transitivity; those are separate queries and bounds.

All three protocol phases were independently verified in ProVerif 2.05. Table 3 summarizes the results; full models are reproduced in Appendices B, C, and D.

Table 3: All five listed ProVerif queries return **TRUE** across the three modeled protocol phases. Session replication is unbounded where the model uses replication; delegation-chain structure remains bounded as stated in §5.3.

Query	Phase	Model	Result
$\neg$ attacker(*master_key*)	v1 (HS256)	`v1_refined.pv`	**TRUE**
Accepted $\implies$ Issued	v1 (HS256)	`v1_refined.pv`	**TRUE**
$\neg$ attacker(*harbor_sk*)	v2 (Ed25519)	`v2_asymmetric.pv`	**TRUE**
Accepted $\implies$ Issued	v2 (Ed25519)	`v2_asymmetric.pv`	**TRUE**
Accepted(b) $\implies$ IssuedRoot(a) $\wedge$ Delegated(a, b)	v3 (Delegation)	`v3_delegation.pv`	**TRUE**

The following is the verbatim ProVerif 2.05 output for the delegation chain query (Phase 3), the most complex property:

```
RESULT event(Accepted(b_2,harbor_id[],cap_1))
  ==> (event(IssuedRoot(a_3,harbor_id[],cap_1))
       && event(Delegated(a_3,b_2,cap_1)))
      || event(IssuedRoot(b_2,harbor_id[],cap_1))
  is true.
```

Listing 1: ProVerif 2.05 Terminal Output — Phase 3 Delegation Chain

3.2 Runtime Enforcement: The Arbiter

Formal models establish properties of their abstractions. At runtime, Port Daddy deploys the **Arbiter**—an ambient monitor that subscribes to the daemon's post-commit activity log and checks six derived invariants:

1. `PID_SQUATTING`: Detects when a process claims a port previously held by a different PID (the "Ghost in the Harbor").
2. `CAP_ESCALATION`: Verifies capability subsets via the deployed Rust FFI enforcer.
3. `NOTE_MONOTONICITY`: Ensures the note sequence for active sessions never shrinks (from the TLA$^+$ `NoteMonotonicity` invariant).
4. `ESCROW_POSITIVE`: Validates that active sessions have positive escrow (when Float Plans are active).

5. `LOCK_OWNER_VALID`: Ensures locks are only held by registered agents with active sessions.
6. `HEARTBEAT_FRESHNESS`: Flags stale heartbeats as early warning for agent death.

Violations are logged and can trigger the "man overboard" salvage protocol: the daemon revokes the offending agent's active Harbor Cards (§2.4), releases any locks and port reservations it held, and, for supervised sessions, notifies the parent agent or the human operator. Because this path observes *committed* events, it detects and compensates; only a pre-commit native guard can prevent a transition in the first place. The Arbiter API provides visibility, not a proof that every security-relevant operation was mediated.

Where the kernel's boundary places this chapter. The controllability theorem of *The Single-Writer Kernel* sorts every rule by whether the event it prohibits is one the mediator can refuse. Card verification is such an event: accept or reject happens on the daemon's own socket, before any effect, so every rule this chapter states about tokens, algorithm pinning, per-hop attenuation, and revocation before admission, is regimentable, and the ProVerif models are proofs about a gate that really can close. What the Anchor cannot regiment is what a holder does with authority it legitimately has. Exercising a valid `db:write` to write the wrong thing is an uncontrollable event at this boundary; it is the Arbiter's detect-and-compensate business here, and audit and slashing in *The Bonded Commons*.

3.3 Implementation Verification

A secure protocol is useless if the implementation contains buffer overflows or timing leaks. We extracted the critical JWT parsing and cryptographic comparison logic into a verified core.

Design invariant 3.2 Memory safety within the harness bound. Using the **Kani** Rust Verifier [21], we bounded-model-check the token parser with key decoding, base64 decoding, and signature verification stubbed to return either outcome. For every 32-byte input that is valid UTF-8 and contains a dot, and within ten loop iterations, the parse path neither panics nor accesses memory out of bounds. The bound is the harness's, not the parser's: longer tokens and the real cryptographic primitives lie outside it.

Design invariant 3.3 Secret-independent source control flow. The Rust comparator (`constant_time_compare`) folds the XOR of every byte pair into one accumulator and tests the accumulator once, so no source-level branch depends on secret bytes. That is a property of the source text, established by inspection; Kani's harness adds only that the comparator cannot panic on any pair of 16-byte inputs. Neither is a hardware-level constant-time proof: compiler transformations, memory behavior, and microarchitecture remain outside both.

3.4 The procedures, precisely

The three phases are restated here as pseudocode, for the reader who wants the exact procedure rather than the narrative. Nothing new is claimed in this subsection: it is the same three phases written so that each line has a visible counterpart in the formal ProVerif models of Appendix B, in the notation used in the protocol-verification literature [1, 5].

3.4.1 Phase 1: Symmetric HS256 with Algorithm Pinning

Algorithm 2 presents the secure verification procedure that is immune to algorithm confusion attacks (CVE-2015-9235 [11]).

```
Require: Pinned Algorithm: hs256
Require: Master Key: k_master (secret)

function Daemon.IssueToken(agent_id, harbor_id, capability):
    jti <- GenerateNonce()
    msg <- Encode(agent_id, harbor_id, jti, capability)
    sigma <- HMAC-SHA256(msg, k_master)
    emit Issued(agent_id, harbor_id, jti, capability)
    return (hs256, msg, sigma)

function Harbor.VerifyToken(tok, expected_harbor):
    (alg_header, msg, sigma) <- tok
    // CRITICAL: Ignore alg_header, pin to HS256
    if HMAC-SHA256(msg, k_master) != sigma:
        return _|_  // Invalid signature
    (a, h, j, cap) <- Decode(msg)
    if h != expected_harbor:
        return _|_  // Wrong harbor
    emit Accepted(a, h, j, cap)
    return (a, h, j, cap)
```

Listing 2: Algorithm 1: Secure Harbor Verification (HS256 with Algorithm Pinning)

Security Property: The algorithm ignores the user-provided alg_{header} (line 15), preventing algorithm confusion attacks where an attacker sets alg = "none" or attempts HS256/RS256 confusion [10].

3.4.2 Phase 2: Asymmetric Ed25519

Algorithm 3 presents the Ed25519-based verification. This phase removes the need for shared HMAC secrets between distributed Harbors.

```
Require: Harbor Keypair: (sk_harbor, pk_harbor)
Require: Key Distribution Channel: c_key (private)

function Daemon.IssueToken(agent_id, harbor_id, capability):
    sk_h <- Receive(c_key)  // Get harbor's secret key
    jti <- GenerateNonce()
    msg <- Encode(agent_id, harbor_id, jti, capability)
    sigma <- Ed25519.Sign(msg, sk_h)
    emit Issued(agent_id, harbor_id, jti, capability)
    return (msg, sigma)

function Harbor.VerifyToken(tok, pk_h, expected_harbor):
    (msg, sigma) <- tok
    if not Ed25519.Verify(msg, sigma, pk_h):
        return _|_  // Invalid signature
    (a, h, j, cap) <- Decode(msg)
    if h != expected_harbor:
        return _|_  // Wrong harbor
    emit Accepted(a, h, j, cap)
    return (a, h, j, cap)
```

Listing 3: Algorithm 2: Asymmetric Harbor Verification (Ed25519)

Security Property: The secrecy of sk_{harbor} and the unforgeability of Ed25519 [8] ensure that only the legitimate Daemon can issue valid Harbor Cards.

3.4.3 Phase 3: Multi-hop Delegation

Algorithm 4 presents the delegation chain verification, inspired by Macaroons [4].

```
Require: Daemon Public Key: pk_daemon
Require: Subset Relation: subseteq on capabilities

function Agent.Delegate(token_parent, sk_parent, agent_child, cap_sub
    ):
    (msg_parent, sigma_parent) <- token_parent
    (_, agent_parent, harbor, cap_parent) <- Decode(msg_parent)
    if cap_sub not_subseteq cap_parent:
        return _|_  // Cannot escalate capabilities
    msg_delegate <- Encode_delegation(msg_parent, sigma_parent,
       agent_child, cap_sub)
    sigma_delegate <- Ed25519.Sign(msg_delegate, sk_parent)
    emit Delegated(agent_parent, agent_child, cap_sub)
    return (msg_delegate, sigma_delegate)

function Harbor.VerifyChain(chain, pk_root):
    tok_root <- chain[0]
    (msg_0, sigma_0) <- tok_root
    if not Ed25519.Verify(msg_0, sigma_0, pk_root):
        return _|_  // Invalid root signature
    cap_prev <- ExtractCaps(msg_0)
    pk_current <- pk_root
    for i <- 1 to |chain| - 1:
        (msg_i, sigma_i) <- chain[i]
        if not Ed25519.Verify(msg_i, sigma_i, pk_current):
            return _|_  // Invalid delegation signature
        cap_i <- ExtractCaps(msg_i)
        if cap_i not_subseteq cap_prev:
            return _|_  // Per-hop capability escalation detected
        cap_prev <- cap_i
        pk_current <- ExtractPubkey(msg_i)
    emit Accepted(agent_final, harbor, cap_prev)
    return T
```

Listing 4: Algorithm 3: Delegation Chain Verification

Security property. Monotonic per-hop attenuation ($\text{cap}_i \subseteq \text{cap}_{i-1}$) prevents capability expansion across transitive delegations, including the `write` → `read` → `write` re-escalation. Property D.1 (§D.1) mechanizes the single hop, where the subset guard is necessary and sufficient. Depth beyond one is the per-hop discipline chained by transitivity of $\subseteq$; its mechanized confirmation at depth two is the attack-and-fix pair in the same section, which checks each hop against its immediate parent rather than against the root.

Recall.

1. Without looking: name the three layers of Table 2, and the two gaps between them that are asserted by a human rather than a tool.
2. What can the Arbiter's post-commit invariants do that ProVerif cannot, and what can they never do that a pre-commit guard could?
3. Which Kani harness does this chapter itself call "a unit test in a proof's clothing," and what property actually rests on the ProVerif model instead?

Exercises 2.1–2.2, page 33.

4 The verified code is the running code

The verified cryptographic core is not a reference implementation—it is the **deployed production code**. The open-source Rust crate `harbor-card-rs` is compiled to a C dynamic library (`cdylib`) and called from the Node.js daemon via FFI through the daemon's runtime-enforcement (Arbiter) module.

This architecture narrows the gap between "verified code" and "running code" that weakens many formal verification efforts: the harnesses are compiled from the same crate the daemon loads at runtime. Kani checks the source under `cfg(kani)`, not the shipped shared library, so the gap that remains is the compiler's.

4.1 Core Verification Logic

```
pub fn verify(&self, token: &str, now_ts: i64)
    -> Result<HarborCardClaims, HarborError> {
    let parts: Vec<&str> = token.split('.').collect();
    if parts.len() != 3 {
        return Err(HarborError::Malformed);
    }
    let msg = format!("{}.{}", parts[0], parts[1]);
    let mut sig_bytes = Self::internal_decode_b64(parts[2])?;
    let sig_result = self.internal_verify_sig(
        msg.as_bytes(), &sig_bytes);
    sig_bytes.zeroize();  // Scrub signature from memory
    sig_result?;
    let mut payload_bytes = Self::internal_decode_b64(parts[1])?;
    let claims: HarborCardClaims =
        serde_json::from_slice(&payload_bytes)?;
    payload_bytes.zeroize();  // Scrub payload from memory
    if claims.exp < now_ts {
        return Err(HarborError::Expired);
    }
    Ok(claims)
}

pub fn constant_time_compare(a: &[u8], b: &[u8]) -> bool {
    if a.len() != b.len() { return false; }
    let mut result = 0u8;
    for i in 0..a.len() { result |= a[i] ^ b[i]; }
    result == 0
}
```

Listing 5: Deployed Rust Implementation — Harbor Card Verifier (abridged from the `harbor-card-rs` crate)

Key implementation details: `zeroize` requests that cryptographic material be overwritten when dropped, reducing residual-secret exposure without proving that no copy survives. The comparator avoids source-level early return on secret byte values. That is useful hygiene, not a hardware constant-time guarantee; compiler, cache, and microarchitectural behavior remain outside this source argument.

4.2 FFI Bridge to the Node.js Daemon

The Rust crate exports two C-ABI functions consumed by the TypeScript daemon:

```
#[no_mangle]
pub extern "C" fn harbor_constant_time_compare(
```

```
    a: *const u8, a_len: usize,
    b: *const u8, b_len: usize
) -> bool { /* ... */ }

#[no_mangle]
pub extern "C" fn harbor_verify_caps_subset_json(
    root_json: *const c_char, root_len: usize,
    sub_json: *const c_char, sub_len: usize
) -> bool { /* ... */ }
```

Listing 6: FFI Exports — Called from the daemon's Arbiter module

The daemon's `arbiter` module binds these at startup:

```
constantTimeCompare: lib.func(
  'bool harbor_constant_time_compare(
     const uint8_t *a, size_t a_len,
     const uint8_t *b, size_t b_len)'),
verifyCapsSubset: lib.func(
  'bool harbor_verify_caps_subset_json(
     const char *root_json, size_t root_len,
     const char *sub_json, size_t sub_len)')
```

Listing 7: TypeScript FFI Binding (daemon-side Arbiter)

4.3 Kani Verification Harnesses

Three bounded model checking proofs are defined in the same source file under `#[cfg(kani)]`:

```
#[cfg(kani)]
#[kani::proof]
#[kani::stub(HarborCardVerifier::internal_pk_from_bytes,
             stubs::pk_from_bytes_stub)]
#[kani::stub(HarborCardVerifier::internal_decode_b64,
             stubs::decode_b64_stub)]
#[kani::stub(HarborCardVerifier::internal_verify_sig,
             stubs::verify_sig_stub)]
#[kani::unwind(10)]
fn proof_verify_logic_only() {
    let pk_bytes: [u8; 32] = kani::any();
    if let Ok(verifier) = HarborCardVerifier::new(pk_bytes) {
        let token_bytes: [u8; 32] = kani::any();
        if let Ok(token_str) =
            std::str::from_utf8(&token_bytes) {
            kani::assume(token_str.contains('.'));
            let _ = verifier.verify(token_str, 0);
        }
    }
}

#[cfg(kani)]
#[kani::proof]
fn proof_constant_time_behavior() {
    let a: [u8; 16] = kani::any();
    let b: [u8; 16] = kani::any();
    let _ = HarborCardVerifier::constant_time_compare(&a, &b);
}

#[cfg(kani)]
```

```rust
#[kani::proof]
fn proof_capability_attenuation() {
    let root = vec!["read".to_string(), "write".to_string()];
    let mut sub = vec!["read".to_string()];
    assert!(HarborCardVerifier::verify_capability_subset(
        &root, &sub));
    sub.push("admin".to_string());
    assert!(!HarborCardVerifier::verify_capability_subset(
        &root, &sub));
}
```

Listing 8: Kani Proof Harnesses (Deployed Source)

The stubbing strategy deserves explanation: `proof_verify_logic_only` stubs out Ed25519 signature verification and base64 decoding (which involve curve arithmetic that Kani cannot symbolically execute within practical bounds) and explores the *parsing and control flow logic*—the split-on-dots, payload extraction, expiry comparison, and error handling paths—for every 32-byte token the harness admits. Within that bound the logic surrounding the cryptographic primitives never panics and never accesses memory out of bounds, whichever outcome the stubbed cryptographic layer returns. The bound is a harness choice, and it has a consequence worth stating: a 32-byte input cannot hold a real three-part card, so what is exercised is the parser's rejection of malformed input, not its acceptance path. The third harness, `proof_capability_attenuation`, checks two concrete vectors and is a unit test in a proof's clothing; the every-hop attenuation property is established by the ProVerif model of §D.1, not by Kani.

The three harnesses run under `cargo kani` in the repository's proof workflow; the run log, not a local build directory, is the evidence that they pass.

Exercises 2.3–2.4, page 33.

5 What the adversary can do, and what it cannot

5.1 Threats this takes seriously

We assume a **Dolev-Yao adversary** [6] who:

- Controls the entire network
- Can intercept, modify, and inject messages
- Cannot break cryptographic primitives (cryptography is perfect)
- May corrupt legitimate agents (but not all simultaneously)

This is the standard threat model for symbolic protocol analysis [1]. Every figure in this chapter that shows a message leaving daemon or verifier authority marks the region this adversary controls with the same dashed band (Figures 2, 3, and 5); figures without a band — the phase progression, the attenuation rings, the cuckoo mechanics, the card lifecycle, the verification stack — show structure inside one trust domain, where the boundary would be decoration rather than a claim.

Table 4 closes the loop on the three failure modes of §1: each is named there, addressed somewhere in between, and carries a residual risk stated here rather than left for the reader to reconstruct.

Table 4: No mechanism in this chapter closes its threat completely: each of the three failure modes of §1 is closed somewhere below, but every row's residual-risk column is deliberately non-empty.

Threat (§1)	Closed where	By what mechanism	Residual risk
Port squatting ("Ghost in the Harbor")	§3, Arbiter invariant `PID_SQUATTING`	Cards bind to the connecting process via the daemon's socket-credential check; the Arbiter flags a port claimed by a PID other than its holder	Detection is post-commit, and the binding is an OS check outside the ProVerif model. See §5.3, "A program is not a person"
Resource contention (shared-state corruption)	Appendix E; SQLite single-writer	One writer per harbor, WAL checkpointing, and `LOCK_OWNER_VALID` in the Arbiter	Enforced by the storage layer, not proved in this chapter; measurable I/O cost
Privilege escalation across delegation	§3.4.3; Property D.1	Per-hop capability attenuation, checked at issuance and re-checked at verification, mechanized against an insider escalation adversary	Mechanized at depth 3 and single-hop for the enriched order; depth is a spawn-time policy check, not a verification-time invariant

5.2 Security Properties

Table 5: The four ProVerif properties hold of the protocol design, not, by themselves, of the shipped code. The three Kani rows are bounded harnesses over the deployed crate: a no-panic check of the parser on 32-byte inputs with the cryptography stubbed, a no-panic check of the comparator on 16-byte pairs, and two concrete vectors for the subset check. "Deployed" indicates the Rust code is called by the production daemon via FFI from the Arbiter module into the `harbor-card-rs` crate.

Property	Verified	Tool	Deployed	Reference
Master Key Secrecy	Yes	ProVerif	—	[1]
Algorithm Confusion Immunity	Yes	ProVerif	—	[10]
Authentication correspondence	Yes	ProVerif	—	[5]
Delegation Integrity	Yes	ProVerif	—	[4]
No-panic parsing (bounded)	Bounded	Kani	Yes (FFI)	[21]
Secret-independent source branching	By inspection	—	Yes (FFI)	[23]
Capability subset check	Two vectors	Kani	Yes (FFI)	[4]

5.3 Where this stops

1. **Delegation-chain depth.** The symbolic proof of delegation soundness (Table 6, "`chain-replay.pv`") is machine-checked at the chain depth realized in deployment (depth 3); it is *not* a proof for unbounded chain length. ProVerif's unbounded-session guarantee quantifies over the number of protocol *sessions*, not over the number of hops in a single delegation chain, which is a fixed structural bound in the modeled process. Extending the proof to unbounded depth (via recursive replication of the delegation process) is future work. The current depth bound is a *spawn-time* policy check (`assessDelegation` in `lib/tube-spawner-router.ts`, `HARD_MAX_DELEGATION_DEPTH` = 8, default 4) applied when a spawn request arrives; it is *not* a verification-time invariant. The cryptographic chain verifier (`lib/delegation-chain.ts`) accepts arbitrary chain depth, and the attenuation monitor (`lib/cap-attenuation-monitor.ts`) enforces only per-hop capability monotonicity—not total depth. Machine-enforced depth at verification time is a planned

hardening item.

2. **A program is not a person.** A Harbor Card authenticates a *capability holder*, not a human and not, by itself, a process. The "Ghost in the Harbor" scenario of §1 is really a PID-binding gap: nothing in the token cryptography stops a *different* process from presenting a card whose original bearer has died, because the card says what may be done, not who is doing it. Port Daddy binds cards to the connecting process outside the token, via the daemon's socket-credential check at connection time (`SO_PEERCRED` on Linux, `LOCAL_PEERCRED`/`getpeereid` on macOS). Two consequences follow, and both are boundaries rather than guarantees. First, this is enforced by the operating system, not proved by ProVerif: the symbolic model has no notion of a process, so a daemon build that skips or mis-orders the check is unprotected regardless of card validity, and no query in Table 3 would notice. Second, the check is only as strong as what it compares. Walk the concrete case: Agent A holds a card and is running as PID 4021. It crashes at t. Three seconds later the OS reuses 4021 for an unrelated Agent B, which presents A's card scavenged from a shared temp file. A credential check that compares *only* the numeric PID admits B — the numbers match. The deployed check therefore compares the peer credential against the socket-connection identity established at card issuance and re-validates process start-time, so a reused PID with a later start-time is rejected; a build that compares the bare PID has an open reuse race. That start-time comparison is a runtime conformance test, not a mechanized proof, and it is listed as such rather than as closed.

3. **`localhost` is a real network.** `localhost` is not a trust boundary. On a shared or multi-user machine any local process — including one belonging to a different user, or a browser page pointed at `127.0.0.1` — can open a TCP connection to a loopback port, and a bearer token sent in the clear over loopback is exactly as interceptable as one sent over the open internet: the Dolev–Yao adversary of §5.1 is not a hypothetical here but the account sharing your build box. Concretely, a card with a 15-minute TTL sniffed off loopback at minute 1 grants its holder fourteen minutes of the victim agent's full capability set, and revocation (§2.4) can shorten that window only after somebody notices. The recommended deployment is a Unix domain socket whose parent directory is mode `0700` and owned by the daemon's own user, so that the filesystem, not the protocol, excludes other users; loopback TLS with a pinned daemon certificate is the alternative where cross-user sharing is genuinely required. The Anchor Protocol does not currently *enforce* either: the daemon will speak to whatever transport it is configured with. This is therefore an operational obligation on the deployer, and one of the few places in this chapter where a wrong configuration silently voids a property the rest of the chapter proves.

Recall.

1. Without looking: state the four properties of the Dolev-Yao adversary this chapter assumes.
2. Of the three failure modes named in §1, which one's residual risk is enforced by the storage layer rather than proved by any tool in this chapter?
3. Why is the PID-reuse fix a runtime conformance test rather than a ProVerif theorem?

Exercise 2.5, page 33.

6 Proofs are leverage, not decoration

A proof is decoration when it is about a model nobody deploys, or when its scope is quietly larger in the prose than in the tool output. It is leverage when it lets you delete a defensive check you would otherwise have had to write, and when the thing proved is the thing that runs. That is the test this chapter has tried to hold itself to: the Kani harnesses are bounded, say

so, and run against the same `harbor-card-rs` crate the daemon loads at runtime; the ProVerif attenuation model was rewritten (Appendix D, §D.1) once a reviewer showed the original could not even *express* the attack it claimed to exclude; and every claim whose mechanization is partial or absent is marked PARTIAL or *open* in Table 6 rather than rounded up.

The Anchor Protocol replaces several hope-based assumptions with explicit, checkable obligations. It combines:

- Symbolic protocol proofs (ProVerif [1])
- Memory-safe implementation (Rust; bounded Kani harnesses [21])
- Capability-based delegation (inspired by Macaroons [4])

Together these artifacts support a bounded assurance case for the modeled and tested surfaces. They do not certify the entire daemon, every delegation depth, or every side channel — and §5.3 says so in the same voice as the claims, because a boundary stated only in a footnote is decoration too.

References

[1] Bruno Blanchet. Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1-2):1–135, 2016. `https://doi.org/10.1561/3300000004` Cited on pages 1, 12, 14, 19, 20, 22, and 31.

[2] Bruno Blanchet. CryptoVerif: A Computationally Sound Security Protocol Verifier. INRIA Research Report, 2007. Cited on page 31.

[3] Bruno Blanchet, Vincent Cheval, and Véronique Cortier. ProVerif with Lemmas, Induction, Fast Subsumption, and Much More. In *IEEE Symposium on Security and Privacy (S&P)*, 2022. Cited on page 31.

[4] Arnar Birgisson, Joe Gibbs Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud. In *Network and Distributed System Security Symposium (NDSS)*, 2014. `https://doi.org/10.14722/ndss.2014.23212` Cited on pages 1, 7, 16, 20, 20, 22, and 32.

[5] Gavin Lowe. A Hierarchy of Authentication Specifications. In *IEEE Computer Security Foundations Workshop (CSFW)*, 1997. Cited on pages 14, 20, and 31.

[6] Danny Dolev and Andrew Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983. Cited on pages 12, 19, and 31.

[7] Jack B. Dennis and Earl C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Communications of the ACM*, 9(3):143–155, 1966. Cited on pages 4 and 32.

[8] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-Speed High-Security Signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012. Cited on pages 7 and 16.

[9] Simon Josefsson and Ilari Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). *RFC 8032*, IETF, January 2017. `https://tools.ietf.org/html/rfc8032` Cited on page 7.

[10] Tim McLean. Critical Vulnerabilities in JSON Web Token Libraries. Auth0 Blog, 2015. `https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/` Cited on pages 15, 20, and 32.

[11] CVE-2015-9235. Algorithm Confusion in jsonwebtoken Node.js library. NIST National Vulnerability Database, 2015. Cited on pages 6, 15, and 32.

[12] CVE-2018-1000531. JWT "none" algorithm vulnerability in jwtf library. NIST National Vulnerability Database, 2018. Cited on page 32.

[13] CVE-2023-48238. Algorithm confusion in json-web-token library. NIST National Vulnerability Database, 2023. Cited on pages 6 and 32.

[14] Yaron Sheffer, Dick Hardt, and Michael Jones. JSON Web Token Best Current Practices. IETF Draft, 2024. `https://tools.ietf.org/html/draft-ietf-oauth-jwt-bcp` Cited on pages 6 and 32.

[15] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975. Cited on page 4.

[16] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated Verification for Secure Messaging Protocols and Their Implementations: A Symbolic and Computational Approach. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017. Cited on page 31.

[17] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A Comprehensive Symbolic Analysis of TLS 1.3. In *ACM Conference on Computer and Communications Security (CCS)*, 2017. Cited on page 31.

[18] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The TAMARIN Prover for the Symbolic Analysis of Security Protocols. In *Computer Aided Verification (CAV)*, 2013. Cited on page 31.

[19] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. A Formal Analysis of 5G Authentication. In *ACM Conference on Computer and Communications Security (CCS)*, 2018. Cited on page 31.

[20] AWS s2n-tls. An Implementation of the TLS/SSL Protocols. `https://github.com/aws/s2n-tls`, 2022. Cited on page 12.

[21] AWS Automated Reasoning Group. Kani Rust Verifier. `https://github.com/model-checking/kani`, 2022. Cited on pages 14, 20, and 22.

[22] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, et al. Everest: Towards a Verified, Drop-in Replacement of HTTPS. In *Logic in Computer Science (LICS)*, 2017. Cited on page 12.

[23] Billy Bob Brumley and Nicola Tuveri. Remote Timing Attacks are Still Practical. In *European Symposium on Research in Computer Security (ESORICS)*, 2011. Cited on page 20.

[24] Manuel Barbosa, Gilles Barthe, Karthikeyan Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-Aided Cryptography. In *IEEE Symposium on Security and Privacy (S&P)*, 2021. Cited on page 31.

[25] Bin Fan, David G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo Filter: Practically Better Than Bloom. In *ACM CoNEXT*, 2014. Cited on page 8.

[26] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic Algorithms for Replicated Database Maintenance. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 1987. Cited on pages 10 and 25.

[27] Universal Commerce Protocol. Specification and reference implementation (Apache-2.0), Google, Shopify, et al., 2026. `https://ucp.dev`. Cited on page 31.

[28] Agent Payments Protocol (AP2): Mandates. Verifiable-credential authorization chain: SD-JWT+kb, JWS detached content, JCS canonicalization, 2026. `https://ap2-protocol.org`. Cited on page 31.

A Formal Verification Status

Artifact availability. Every artifact named in Table 6 is bundled at `https://portdaddy.dev/whitepaper/artifacts/`. The verified Rust core is the open-source `harbor-card-rs` crate; runtime components are deployed inside the Port Daddy daemon binary. *The Bonded Commons* carries the cross-paper status table; this appendix lists only the items that pertain specifically to the Anchor Protocol.

Table 6: Six of the twelve listed claims close at both the model and runtime level; five remain partial and one open. **closed**: model verified *and* runtime conformance test passes. PARTIAL: design verified, runtime empirically tested rather than proved. *open*: known unmechanized.

Claim	Method	Result	Artifact
Authentication correspondence in the listed phase models (§3.4)	ProVerif 2.05	**closed** non-injective correspondence queries TRUE	`harbor_card_v*.pv`
Algorithm-confusion immunity	ProVerif pinned vs. naive	**closed** pinned TRUE; counter-trace witnessed	`algconfusion.pv`
Delegation chain replay (§3.4.3)	ProVerif at depth 3	**closed** chain authenticity TRUE; 17-case runtime conformance	`chain-replay.pv`
Cuckoo-filter pollution resistance (§2.4)	5-invariant property test	**closed** FP rate within $2B/2^f$ at load 0.95	cuckoo property tests
GitHub-App webhook origin authentication	ProVerif (Dolev-Yao relay)	**closed** origin auth TRUE; daemon re-verifies GitHub HMAC (runtime conformance)	`ingress_origin_auth.pv`
Multi-tenant relay namespace isolation	ProVerif + runtime	**closed** member of A cannot publish into B; `HARBOR_MISMATCH` conformance	`ingress_tenant_isolation.pv`
Cuckoo-filter revocation soundness	inspection + empirical	PARTIAL gate narrows acceptance; dissemination remains assumption-bound	runtime test
Constant-time signature comparison	audited `subtle::ConstantTimeEq` library	PARTIAL no side-channel proof; library trust documented	—
Relay payload secrecy via HPKE (daemon key pinned)	ProVerif (design)	PARTIAL secrecy TRUE under pinned key; seal not yet implemented	`ingress_hpke_secrecy.pv`
Webhook replay-freedom + in-order delivery	TLA+ / TLC (exhaustive at bound)	PARTIAL design model-checked; runtime via ordered-chain ingest	`WebhookDelivery.tla`

(continued)

Claim	Method	Result	Artifact
Card revocation finality under backup/restore (ADR-0018 Attack 2)	TLA+ / TLC	PARTIAL rollback attack + revocation-epoch mitigation both model-checked	`CardRevocation.tla`
Mechanized gossip-convergence bound	—	*open* standard expected asymptotics only [26]; no partition deadline	—

Limitations. The transition to gossiping the Append-Only Event Log removes the unsound filter merge described in §2.4 ("Why filters cannot be merged bitwise"), but cross-peer gossip propagation still leans on the standard anti-entropy analysis rather than a mechanized proof. The side-channel resistance of signature comparison is inherited from an audited library rather than re-proved. These deferrals are documented as known open problems rather than as defects in the present claims.

GitHub-App relay ingress. The webhook-dispatch surface (GitHub → untrusted relay → daemon) is modeled with the relay as the Dolev-Yao attacker, in three relay-agnostic properties, each paired with a negative control (`*_vuln.pv`) that exhibits the attack once the mitigation is removed—so a *true* result is non-vacuous. (i) *Origin authentication*: the daemon dispatches a fleet event only if GitHub signed it, even when the forwarder's transport credential is attacker-held; the control omits the daemon-side HMAC re-verification and the property fails (forged dispatch). This is *non-injective* agreement (origin), not replay-freedom: a captured genuine (*payload*, *mac*) may be re-delivered. Replay-freedom is an ordering-sensitive, mutable-state property outside ProVerif's reach, so it is discharged separately in TLA+ (`proofs/relay/WebhookDelivery.tla`): a daemon consuming the *ordered* per-publisher chain (dispatch iff $seq = tip + 1$) is model-checked at-most-once *and* gap-free against an adversarial relay that reorders, duplicates, and redelivers, with the unguarded daemon as the model-checked counterexample (TLC exhibits a double-dispatch). The obligation it names: the daemon ingests webhooks over the relay's ordered `from_seq` subscribe path, not the raw forward. (ii) *Payload secrecy*: under HPKE to the daemon's *pinned* X25519 key the relay never learns plaintext; the control sources the key from the relay and the secrecy property collapses to a key-substitution MITM—hence out-of-band key pinning is a requirement, not a recommendation. The seal is specified but not yet implemented, so this row is PARTIAL. (iii) *Namespace isolation*: a member of tenant A cannot publish into tenant B because harbor binding is checked against authoritative membership; the control trusts the card's self-declared issuer and crosses the tenant boundary. Two limits: HMAC-SHA256 gives origin authentication to a shared-key holder (EUF-CMA), not third-party non-repudiation; HPKE base mode is assumed IND-CCA2.

> *We err toward listing fewer items as closed rather than more.*

B Full ProVerif Models

For completeness, we include the full ProVerif models used for verification. These correspond to the algorithms in Section 3.4.

B.1 Phase 1: Symmetric HS256

```
(* Port Daddy: Harbor Card v1 (HS256) Formal Model *)
```

```
type id. type key. type jti. type capability. type alg_type.
const hs256_alg: alg_type. const none_alg: alg_type.
free c: channel.

fun hs256(bitstring, key): bitstring.
reduc forall m: bitstring, k: key;
  check_hs256(m, k, hs256(m, k)) = true.

reduc forall m: bitstring, k: key;
  verify_generic(m, hs256_alg, hs256(m, k), k) = true;
  forall m: bitstring, k: key;
  verify_generic(m, none_alg, m, k) = true.

fun encode_token(id, id, jti, capability): bitstring [data].

event Issued(id, id, jti, capability).
event Accepted(id, id, jti, capability).

free master_key: key [private].
query attacker(master_key).

query a: id, h: id, j: jti, cap: capability;
  event(Accepted(a, h, j, cap)) ==> event(Issued(a, h, j, cap)).

let Daemon(k: key, a_id: id, h_id: id, j: jti, cap: capability) =
  let msg = encode_token(a_id, h_id, j, cap) in
  let signature = hs256(msg, k) in
  event Issued(a_id, h_id, j, cap);
  out(c, (hs256_alg, msg, signature)).

let SecureHarbor(k: key, expected_h: id) =
  in(c, (alg_header: alg_type, msg: bitstring, signature: bitstring))
     ;
  if check_hs256(msg, k, signature) = true then
    let encode_token(a: id, h: id, j: jti, cap: capability) = msg in
    if h = expected_h then
      event Accepted(a, h, j, cap).

process
  new agent_id: id; new harbor_id: id;
  new token_jti: jti; new secret_cap: capability;
  (!Daemon(master_key, agent_id, harbor_id, token_jti, secret_cap) |
   !SecureHarbor(master_key, harbor_id))
```

Listing 9: harbor_card_v1_refined.pv

C Phase 2: Asymmetric Ed25519

```
(* Port Daddy: Harbor Card v2 (Ed25519 / EdDSA) Formal Model *)
type id. type skey. type pkey. type jti. type capability.

free c: channel.
free k_dist: channel [private]. (* Trusted key distribution *)

(*** Digital Signatures (Ed25519) ***)
fun pk(skey): pkey.
fun sign(bitstring, skey): bitstring.
```

```
reduc forall m: bitstring, k: skey;
  check_sign(m, pk(k), sign(m, k)) = true.

fun encode_token(id, id, jti, capability): bitstring [data].

(*** Events ***)
event Issued(id, id, jti, capability).
event Accepted(id, id, jti, capability).

(*** Queries ***)
free secret_key: skey [private].
query attacker(secret_key).

query a: id, h: id, j: jti, cap: capability;
  event(Accepted(a, h, j, cap)) ==> event(Issued(a, h, j, cap)).

(*** Processes ***)
let Daemon(a_id: id, h_id: id, j: jti, cap: capability) =
  in(k_dist, sk_h: skey);
  let msg = encode_token(a_id, h_id, j, cap) in
  let signature = sign(msg, sk_h) in
  event Issued(a_id, h_id, j, cap);
  out(c, (msg, signature)).

let Harbor(pk_h: pkey, expected_h: id) =
  in(c, (msg: bitstring, signature: bitstring));
  if check_sign(msg, pk_h, signature) = true then
    let encode_token(a: id, h: id, j_1: jti, cap_1: capability)
      = msg in
    if h = expected_h then
      event Accepted(a, h, j_1, cap_1).

(*** Main Process ***)
process
  new agent_id: id; new harbor_id: id;
  new token_jti: jti; new secret_cap: capability;
  let harbor_pk = pk(secret_key) in
  out(c, harbor_pk);
  (
    (!Daemon(agent_id, harbor_id, token_jti, secret_cap)) |
    (!Harbor(harbor_pk, harbor_id)) |
    (!out(k_dist, secret_key))
  )
```

Listing 10: `harbor_card_v2_asymmetric.pv` — Verified in ProVerif 2.05

D Phase 3: Multi-hop Delegation

```
(* Port Daddy: Harbor Card v3 (Delegation Chains) Formal Model *)
type id. type skey. type pkey. type capability.

free c: channel.

(*** Cryptographic Functions ***)
fun pk(skey): pkey.
fun sign(bitstring, skey): bitstring.
reduc forall m: bitstring, k: skey;
```

```
  check_sign(m, pk(k), sign(m, k)) = true.

reduc forall c1: capability; is_subset(c1, c1) = true.

fun base_token(id, id, id, capability): bitstring [data].
fun delegate_token(bitstring, id, capability): bitstring [data].

(*** Events ***)
event IssuedRoot(id, id, capability).
event Delegated(id, id, capability).
event Accepted(id, id, capability).

(*** Queries ***)
free harbor_id: id.
query b: id, cap: capability, a: id;
  event(Accepted(b, harbor_id, cap)) ==>
    (event(IssuedRoot(a, harbor_id, cap))
     && event(Delegated(a, b, cap)))
    || event(IssuedRoot(b, harbor_id, cap)).

(*** Processes ***)
free daemon_id: id.
free daemon_sk: skey [private].

let Daemon(a: id, cap: capability) =
  let msg = base_token(daemon_id, a, harbor_id, cap) in
  let s = sign(msg, daemon_sk) in
  event IssuedRoot(a, harbor_id, cap);
  out(c, (msg, s)).

let AgentA(a: id, a_sk: skey, b: id, sub_cap: capability) =
  in(c, (msg: bitstring, s: bitstring));
  let base_token(=daemon_id, =a, =harbor_id,
    root_cap: capability) = msg in
  if check_sign(msg, pk(daemon_sk), s) = true then
    if is_subset(sub_cap, root_cap) = true then
      let d_msg = delegate_token((msg, s), b, sub_cap) in
      let d_s = sign(d_msg, a_sk) in
      event Delegated(a, b, sub_cap);
      out(c, (d_msg, d_s)).

let Harbor(a_pk: pkey) =
  in(c, (d_msg: bitstring, d_s: bitstring));
  let delegate_token((base_msg: bitstring, base_s: bitstring),
    b: id, sub_cap: capability) = d_msg in
  let base_token(=daemon_id, a: id, =harbor_id,
    root_cap: capability) = base_msg in
  if check_sign(base_msg, pk(daemon_sk), base_s) = true then
  if check_sign(d_msg, a_pk, d_s) = true then
  if is_subset(sub_cap, root_cap) = true then
    event Accepted(b, harbor_id, sub_cap).

(*** Main Process ***)
process
  new sk_a: skey;
  let pk_a = pk(sk_a) in
  out(c, pk_a);
  new id_a: id; new id_b: id; new cap_root: capability;
```

```
68    (
69      (!Daemon(id_a, cap_root)) |
70      (!AgentA(id_a, sk_a, id_b, cap_root)) |
71      (!Harbor(pk_a))
72    )
```

Listing 11: `harbor_card_v3_delegation.pv` — reflexive-order model (see §D.1 for the soundness caveat and the enriched v5 proof)

D.1 Soundness of the Attenuation Proof (v5)

The v3 model above is faithful to the cryptographic structure but *vacuous about attenuation*. Its capability order is reflexive only:

```
reduc forall c1:  capability; is_subset(c1, c1) = true.
```

Capabilities are opaque atoms with no order, so a delegated capability can only ever *equal* its parent. Escalation—delegating more authority than you hold—is not blocked; it is *inexpressible*. The no-escalation query therefore holds for a reason that has nothing to do with the protocol: there is no larger capability to delegate. A reviewer flagged this directly: *"the model cannot witness escalation."*

`harbor_card_v5_attenuation.pv` closes the obligation by enrichment, not by weakening the claim. It gives capabilities a real partial order (`cap_read` $\sqsubset$ `cap_write`, both public so the attacker can construct either), replaces the reflexive stub with the sound order (`is_subset(cap_read, cap_write)` derivable; `is_subset(cap_write, cap_read)` *not*), and adds an *insider escalation adversary*: an agent holding a valid `cap_read` root token that signs an over-broad `cap_write` delegation with no self-restraint. The verifier's `is_subset` guard is the sole defense under test.

```
1  (* Sound order: reflexive + read <= write; write <= read NOT
       derivable *)
2  fun is_subset(capability, capability): bool
3  reduc
4      forall x: capability; is_subset(x, x) = true
5      otherwise is_subset(cap_read, cap_write) = true.
6
7  (* Insider adversary: holds a real root, signs an over-broad
       delegation *)
8  let MaliciousA(a: id, a_sk: skey, b: id) =
9    in(c, (msg: bitstring, s: bitstring));
10   let base_token(=daemon_id, =a, =harbor_id, root_cap: capability) =
         msg in
11   if check_sign(msg, pk(daemon_sk), s) = true then
12     event EscalationAttempted(a, root_cap, cap_write);
13     let d_msg = delegate_token((msg, s), b, cap_write) in
14     out(c, (d_msg, sign(d_msg, a_sk))).
15
16 (* Q1 soundness: a write-card delegated from a read-root is NEVER
       accepted *)
17 query b: id; event(Accepted(b, harbor_id, cap_write, cap_read)) ==>
       false.
18 (* Q2 non-vacuity: the escalation IS reachable (the proof is not
       vacuous) *)
19 query a: id, r: capability, s: capability;
20   event(EscalationAttempted(a, r, s)) ==> false.
```

Listing 12: `harbor_card_v5_attenuation.pv` (excerpt) — Verified in ProVerif 2.05

Model-checked property D.1 Attenuation soundness, mechanized. In `harbor_card_v5_attenuation.` ProVerif 2.05 reports `Accepted(b, harbor, cap_write, cap_read)` unreachable (**Q1 = true**): no escalating delegation is ever accepted. Simultaneously `EscalationAttempted` is reachable (**Q2 = false-with-trace**): the attack is expressible, so Q1 is a real defense, not the v3 vacuum. A negative control that deletes the verifier's `is_subset` guard flips Q1 to false-with-trace—ProVerif exhibits the read→write escalation— confirming the guard is necessary.

The property was not always true of the model. The version-6 model let a delegation name a capability its parent did not hold, and ProVerif produced the trace; version 7 adds the per-hop subset check and the same query closes. The two verdicts, from the committed results files:

At the terminal. *the multi-hop attack on version 6, and version 7 closing it.*

```
$ proverif analyses/harbor_card_v6_multihop_attack.pv
-- Query not event(Accepted(cc_2,harbor_id[],cap_write)) in process 1.
goal reachable: event(Accepted(id_c[],harbor_id[],cap_write))

  ... derivation, 11 steps, elided; the trace is in analyses/harbor_card_v6_results.txt ...

A trace has been found.
RESULT not event(Accepted(cc_2,harbor_id[],cap_write)) is false.

$ proverif analyses/harbor_card_v7_multihop_fixed.pv
RESULT not event(Accepted(cc_2,harbor_id[],cap_write)) is true.
```

Multi-hop scope. Property D.1 is single-hop (Daemon → A → verifier). Multi-hop delegation has a sharper obligation: a verifier that checks the *final* capability against the *root* accepts a non-monotonic middle hop. `harbor_card_v6_multihop_attack.pv` mechanizes exactly this — A:`write` → B:`read` → C:`write` is accepted (ProVerif: escalation reachable). The fix, `harbor_card_v7_multihop_fixed.pv`, verifies each hop against its *immediate parent* (is_subset(*final*, *mid*) ∧ is_subset(*mid*, *root*)) and ProVerif proves the same chain rejected. The runtime delegation verifier — and the cross-harbor recompute $\text{att}_B \circ \text{att}_A$ — must be per-hop, never final-vs-root.

Exercises 2.6–2.9, page 33.

E Limitations & Boundaries

The formal mechanisms presented in this chapter are mathematically sound within their defined scopes, but they depend on bounding assumptions that must be explicitly acknowledged.

- **Threat Model Exclusions:** Collusion across all independent organs (e.g., the logging daemon, the arbitrator, and the primary execution runtime) is assumed out-of-scope; if all enforcing components are compromised, the guarantees degrade to advisory.
- **Performance Trade-offs:** Cryptographic assertions and WAL-checkpointing for per-write durability incur measurable I/O overhead. These mechanisms are designed for high-stakes coordination, not for bulk data processing or high-frequency trading.
- **Implementation Maturity:** While the core protocol (Harbor Cards, delegation, revocation) is IMPLEMENTED and running in production, the unbounded-depth delegation proof,

machine-enforced verification-time chain-depth limits, and mandatory transport-boundary enforcement discussed in §5.3 remain PARTIAL or DESIGNED.

These constraints are not flaws but structural realities of building zero-trust coordination on top of POSIX primitives.

F Further reading and prior art

This appendix traces the lineage the main text points at in §1. It is placed here rather than between the protocol description and the verification argument so that the chapter's narrative thread is not cut in half by a literature review; nothing in §§2–6 depends on reading it. The most immediately consequential subsection — the credential-format migration now under way — comes first.

F.1 Credential Formats in Agentic Commerce

Since this protocol was first specified, the agentic-payments industry has converged on a concrete credential dialect: the Agent Payments Protocol [28], adopted by the Universal Commerce Protocol [27] (Google/Shopify, 2026), carries its authorization mandates as W3C Verifiable Credentials in SD-JWT form — SD-JWT+kb for principal proof, JWS detached-content signatures (RFC 7515 Appendix F) over JCS-canonicalized payloads (RFC 8785) for counterparty authorization, with ES256 as the recommended algorithm and keys published as JWK sets in a `/.well-known` profile. The reference implementation migrates the Harbor Card envelope to this profile (`hv: 3`, ADR-0094) so that external verifiers need no bespoke SDK. The migration is deliberately envelope-only: the delegation-chain semantics, the attenuation invariant of Property 3.1, and the ProVerif obligations are independent of the serialization, though the key-binding construction of SD-JWT+kb adds a binding obligation the model must re-discharge. The JWT hardening below (strict algorithm whitelisting, key separation) transfers unchanged — SD-JWT is a JWT profile, not a departure from one. The chapter's keywords name the protocol's JWT lineage; SD-JWT is where that lineage currently lands, not a departure from it.

F.2 Formal Verification of Security Protocols

Formal verification of cryptographic protocols has matured significantly over the past two decades. Following the seminal work by Lowe [5] on authentication hierarchies and the Dolev-Yao model [6], several automated tools have been developed:

- **ProVerif** [1]: An automatic symbolic protocol verifier supporting an unbounded number of sessions, used to verify TLS 1.3 [17], Signal [16], and WireGuard. The 2.05 series used here adds the lemma, induction, and subsumption machinery described by Blanchet et al. [3].
- **Tamarin** [18]: A state-of-the-art protocol verifier that supports equational properties and inductive proofs, used for analyzing 5G authentication [19].
- **CryptoVerif** [2]: A computationally-sound protocol verifier that provides proofs in the computational model rather than the symbolic model.

Our work builds on these foundations, applying established verification techniques to the novel domain of local multi-agent coordination. Barbosa et al. [24] survey the wider computer-aided-cryptography landscape and, in particular, the gap between symbolic and computational guarantees that §5.3 declines to paper over.

F.3 Capability-Based Authorization

Capability-based security dates back to Dennis and Van Horn's seminal 1966 paper on programming semantics for multiprogrammed computations [7]. Recent work by Birgisson et al. [4] introduced Macaroons, which use chained HMACs for decentralized delegation with contextual caveats. The Anchor Protocol's delegation chains (§3.4.3) are inspired by Macaroons but adapted for JWT-based identity in local development environments.

F.4 JWT Security

JSON Web Tokens have been the subject of extensive security analysis. Algorithm confusion attacks were first systematically studied by McLean [10], with subsequent vulnerabilities documented in CVE-2015-9235 [11] (HS256/RS256 confusion), CVE-2018-1000531 [12] ("none" algorithm), and CVE-2023-48238 [13] (generic algorithm confusion). Our protocol implements the defense recommended by the IETF JWT Best Current Practices [14]: strict algorithm whitelisting and key separation.

Review of the key ideas

What this chapter leaves coupled, in the order the rest of the book will lean on it:

1. A **harbor card** is a short-lived signed statement of exactly what one agent may do; the daemon checks the card, not the process, and a process with no card can do nothing the harbor mediates.
2. **Attenuation is monotone**: a child card is a subset of its parent in scope and in lifetime, at every hop, and the chain's task hash is the same at every hop. This is the property ProVerif checks on the multi-hop model, and it is the one the version 6 model was found to lack.
3. **Algorithmic pinning**: the verifier chooses the signature algorithm from policy, never from the token's header. The same bytes that a header-selected verifier accepts, a policy-pinned verifier refuses.
4. **Revocation is an epoch, not a message**: a revoked card leaves the admissible set only when the filter for its epoch reaches the daemon, so there is a stale window of at most two gossip intervals in a connected fleet and no finite bound under partition.
5. **Filters do not merge; logs do.** Two daemons cannot union their cuckoo filters, so the append-only revocation log is the shared object and each daemon rebuilds its own filter from it.
6. **Correction is a new card**, with a new identifier and a fresh authority; the old card's history is untouched, and nothing is reanimated in part.
7. The proofs are **bounded and named**: authentication correspondence for the modeled phase, attenuation soundness on the multi-hop model, memory safety within the harness bound with the cryptography stubbed, and branch-freedom at source level. Each is a statement about a model, and the chapter says which.
8. What the adversary can still do is stated as scope: a process that holds a valid card can spend it in full within its lifetime; the protocol bounds what a card grants, not what a grantee wants.

G Exercises

§3 How we know it is correct

2.1 Trace ⋆⋆ Read `proofs/anchor/token-verify/algconfusion.pv`. State the algorithm-confusion attack in words: what the attacker already knows, what it forges, and which verifier admits it. Then say, in one sentence, what *pinning* must mean for a verifier to resist it. *(solution on page 34)*

2.2 Check ⋆ `proofs/anchor/delegation/chain-replay.pv` models a 3-hop chain $P \to A \to B \to C$. Which property does the model check, and what does the run log's verdict say? *(solution on page 34)*

§4 The verified code is the running code

2.3 Check ⋆⋆ For each of the three Kani harnesses above, state exactly what is proved and name one stronger claim a reader might wrongly draw from it. *(solution on page 34)*

2.4 Open ⋆⋆⋆ None of this chapter's ProVerif models, and none of its three Kani harnesses, says anything about most of what the running daemon does at a socket. Using only what this chapter already states about the Kani harnesses' scope, write the one-paragraph boundary a reader must hold before trusting "verified" as a description of the deployed binary. Then say what a stronger harness would need to add. *(solution on page 35)*

§5 What the adversary can do, and what it cannot

2.5 Trace ⋆⋆ A Harbor Card verifies: `Ed25519.Verify` returns true. Trace what else must hold before that verified signature actually *authorizes* the action its bearer is attempting. Then say what changes if the daemon treats the token as a bearer credential rather than re-checking it on every request. *(solution on page 35)*

§D Phase 3: Multi-hop Delegation

2.6 Trace ⋆⋆ Trace the chain A:`write` $\to$ B:`read` $\to$ C:`write` through two verifiers by hand: (i) one that checks only the final capability against the root, and (ii) the per-hop verifier of Algorithm 4. Which one accepts C? Then say precisely why checking every $\text{cap}_i \subseteq \text{cap}_{i-1}$ proves $\text{cap}_k \subseteq \cdots \subseteq \text{cap}_0$, where checking only $\text{cap}_i \subseteq \text{cap}_0$ at each hop does not. *(solution on page 36)*

2.7 Trace ⋆⋆ Read `analyses/harbor_card_v6_multihop_attack.pv`. State the attack in words: who forges what, at which hop, and why does a verifier that checks only the final capability against the root admit it? *(solution on page 36)*

2.8 Check ⋆ `analyses/harbor_card_v7_multihop_fixed.pv` is identical to v6 except for the verifier. Name the one modeling change that closes the attack, and what the RESULT line becomes. *(solution on page 36)*

2.9 Trace ⋆⋆⋆ The delegation chains in this chapter use signed Ed25519 cards. `analyses/macaroon_discharge_v1.pv` and `analyses/macaroon_discharge_v2_naive_unsound.pv` model a related but different construction, an HMAC-chained macaroon with a third-party discharge caveat. Compare the two: what does the naive discharge verifier in v2 admit, and what does v1's verifier check that closes it? *(solution on page 36)*

Solutions to the exercises

2.1 *(exercise on page 33)* The model hands the attacker *both* the honest issuer's public key `vkA` and the HMAC key `vkB` in the clear — the worst case for a verifier that is only supposed to trust Ed25519-shaped tokens. `VerifierNaive` does not pin an algorithm; it pattern-matches on whichever type tag arrives, `tokenA` or `tokenB`, and calls that tag's own verification routine. Knowing `vkB`, the attacker builds `tokenB(ALG_B, m, hmacB(vkB, m))` for any message `m`, including one the honest `IssuerA` never signed; the naive verifier's second branch matches the `tokenB` shape, the HMAC checks out under a key the attacker already holds, and `accepted_naive(m)` fires with no matching `issued_A(m)`. `VerifierPinnedA`, by contrast, only ever destructures the `tokenA` constructor — a `tokenB` token is not merely rejected, it is a shape the pinned verifier never attempts to read. Pinning means fixing, out of band, which single verification routine and key a given context accepts, so the token's own header or shape can never be the thing that chooses how it gets checked — exactly the rule of Design invariant 2.1, mechanized here against an attacker who already holds every key the naive path would trust.

Pinned verdict (`proofs/anchor/token-verify/algconfusion.run.log`):

```
RESULT event(accepted_pinned(m_4)) ==> event(issued_A(m_4)) is true.
RESULT event(accepted_naive(m_4)) ==> event(issued_A(m_4)) is false.
```

The first line is the property this chapter relies on; the second is the negative control, checked in the same file so the true result is not vacuous.

2.2 *(exercise on page 33)* It checks that every chain C accepts corresponds to one P actually authorized, with the *same* (P, A, B, C, m) tuple: no hop can be replayed against a different message, and no hop can be spliced out of one chain into another. The defense is per-hop nonce freshness — each signature binds `hopBind(nonce, prev_id, next_id, message)`, and the verifier requires every nonce to have been issued and consumes it on acceptance, so a captured hop cannot be re-presented in a different context.

Pinned verdict (`proofs/anchor/delegation/chain-replay.run.log`):

```
RESULT event(chain_accepted(idP_4,idA_3,idB_2,idC_1,m_4)) ==> event(
    chain_authorized(idP_4,idA_3,idB_2,idC_1,m_4)) is true.
```

This is a different property from Property D.1's capability-subset soundness: chain replay says who signed what is delivered intact; attenuation soundness says authority never grows along the way. §5.3 cites this exact artifact as the depth-3 evidence behind the "Delegation-chain depth" boundary.

2.3 *(exercise on page 33)* `proof_verify_logic_only` stubs signature verification and base64 decoding and shows the surrounding parse and branch logic cannot panic on any 32-byte, dot-containing, UTF-8 input within ten unwindings — a bounded absence-of-panic result under those stubs, not a proof that `verify` itself is correct. A 32-byte input cannot even hold a real three-part card, so only the *rejection* path is exercised; a reader should not conclude the acceptance path was checked at all.

`proof_constant_time_behavior` calls the comparator once over symbolic 16-byte pairs and shows it cannot panic; it is not a two-run relational proof that the *time* taken is independent of the secret, and Kani adds nothing here beyond the source-level argument already given by inspection (no early return on a secret byte). Compiler, cache, and microarchitectural timing stay entirely outside it — a reader should not read "Kani checked it" as "this is constant-time in the deployed binary."

`proof_capability_attenuation` asserts two concrete, hard-coded vectors, one subset accepted and one escalation rejected; it is a unit test wearing a Kani harness, not a universal statement over arbitrary capability structures. The actual every-hop attenuation property is

what §D.1's ProVerif model establishes, not this harness — a reader should not treat the two vectors as covering the property Property D.1 states.

The bound in each case is recorded, not just asserted here: `whitepaper/corpus.json` carries an `evidencePolicy` field for `harbor-card-kani-verify-logic-only`, `harbor-card-kani-constant-time`, and `harbor-card-kani-capability-attenuation` stating exactly these limits, so a claim that drifted past this chapter's wording would be visible against the manifest rather than only against memory.

2.4 *(exercise on page 33)* As this chapter states it, not more: the Kani harnesses give bounded absence-of-panic for the token parser on 32-byte inputs with signature verification and base64 decoding stubbed out entirely, a source-level argument that the comparator's branches do not depend on secret bytes, and two concrete, hard-coded vectors for the capability-subset check. None of the three is a proof about the real cryptography (stubbed away in the first, untouched by the other two), about inputs larger than the harness's own bound, about the compiled binary's timing behavior, about the FFI boundary itself, or about any code path outside `harbor-card-rs`. The ProVerif models are symbolic proofs about the *protocol* under a Dolev-Yao adversary, not about this Rust source at all — the bridge between "the model" and "this source" is exactly the assumption gap Table 2 lists in its last column, asserted by a human, not discharged by either tool.

A stronger harness would replace the two hard-coded attenuation vectors with an arbitrary capability structure under an asserted partial-order property, so the check is over the order rather than two fixed cases, with a negative control that fails when the `is_subset` guard is removed — the same discipline the ProVerif attenuation model already applies for Property D.1. It would also need to state parser refinement (that the stubbed primitives match their real implementations) as a separate, currently absent, obligation. None of that exists yet, and this chapter does not claim it does: `whitepaper/corpus.json` pins the three harnesses' present bounds in their own `evidencePolicy` fields, and `scripts/proofs/run-proverif.py` (CI job "ProVerif / model estate") re-runs every ProVerif model named in this chapter's exercises against its committed source on every change — the freshness guarantee behind every pinned line quoted above, not a claim about the harnesses it does not mechanize.

2.5 *(exercise on page 33)* A valid signature proves exactly one thing: the holder of the signing key produced these bytes. It says nothing, by itself, about canonical decoding (did the verifier parse the *same* bytes the signer meant, under one canonical encoding); key-to-principal binding (whose key is this, and is that binding itself authenticated — §5.1's PID-reuse case is a binding failure downstream of a perfectly valid signature); audience and resource semantics (was this token issued for *this* harbor and *this* action); freshness (`exp` checked against a clock the bearer does not control); ancestor revocation (§2.4 — a signature does not know its own chain was withdrawn upstream); or that every consequential effect actually consults the verifier at all. Table 5 lists only the cryptographic set as ProVerif-verified; authorization is the larger claim built on top of it, not a synonym for it.

Because Harbor Cards are bearer credentials — whoever presents the bytes gets the capability, with no further proof of possession — a card sniffed off loopback (§5.3, "`localhost` is a real network") is exactly as usable to an attacker as to its intended holder for the remainder of its TTL. Re-checking per request narrows the window in which a *revoked* card is still honored (§2.4); it does nothing against a card that is merely stolen and still technically valid, because the verifier cannot distinguish the two bearers by the token alone.

A status table sharper than Table 6's three tiers would keep four things distinct rather than folding them into one word: symbolic protocol correspondence at a fixed modeled hop depth (what ProVerif proved); bounded implementation memory and control-flow checks (what Kani proved); runtime verifier integration (whether the deployed path actually calls the verified routine on every effect); and system-level complete mediation (whether every effect-producing

path is reachable only through that call). This chapter's artifacts presently evidence the first two; the last two are asserted in prose (§4, "narrows the gap … to the compiler's") rather than mechanized, and reading any result in this chapter as more than it is means collapsing exactly this distinction.

2.6 *(exercise on page 33)* (i) A root-vs-final verifier asks only `is_subset(cap_write, cap_root)`: since the root itself holds write, this is true, and C is accepted with write — an authority B never held and never legitimately re-delegated. (ii) Algorithm 4's loop checks each hop against the one immediately before it: hop 1, `is_subset(cap_read, cap_root)`, holds, so B's grant stands; hop 2, `is_subset(cap_write, cap_read)`, fails, because write is not inside what B was actually handed, so C is rejected at the exact hop where authority tries to grow back. Only the root-vs-final verifier accepts; the per-hop verifier does not.

The induction: if every hop satisfies $\text{cap}_i \subseteq \text{cap}_{i-1}$, transitivity of $\subseteq$ chains them directly into $\text{cap}_k \subseteq \text{cap}_{k-1} \subseteq \cdots \subseteq \text{cap}_0$. Checking $\text{cap}_i \subseteq \text{cap}_0$ independently at each hop supplies no such chain: it only asks whether each capability fits inside the *root*, which a re-escalating middle hop can satisfy (write $\subseteq$ root) while still expanding relative to what that hop's own parent actually held (write $\not\subseteq$ B's read). This is why the sentence following Algorithm 4 credits Property D.1 with the single hop only: the multi-hop induction is what the per-hop *discipline* buys structurally, and its mechanized confirmation is the v6/v7 pair immediately below, not a restatement of the property at depth greater than one.

2.7 *(exercise on page 33)* The root holds `cap_write`. Honest A correctly attenuates to `cap_read` for B — a legitimate restriction. Malicious B, holding only that read-capped delegation, signs a *new* delegation to C carrying `cap_write`: an authority B never held, forged with B's own legitimately-issued delegation key, not by breaking any signature. `NaiveVerifier` unwraps the two-hop chain and checks `is_subset(final_cap, root_cap)` — the final capability against the root only — and never re-derives what the intermediate hop actually held. Because `cap_write` is a subset of the root, the check passes and C is accepted with full write authority, although B itself only ever held read. This is hop-2 escalation: a child re-grants an authority its own parent narrowed away, and a root-vs-final comparison is structurally blind to it.

Pinned output (`analyses/harbor_card_v6_results.txt`):

```
1 The event Accepted(id_c_1,harbor_id,cap_write) is executed at {43} in
      copy a_10.
2 A trace has been found.
3 RESULT not event(Accepted(cc_2,harbor_id[],cap_write)) is false.
```

ProVerif does not merely report the query false; it exhibits the concrete trace above as a witness.

2.8 *(exercise on page 33)* The only change is in the verifier: v6's single check `is_subset(final_cap, root_cap)` becomes two checks in sequence, `is_subset(mid_cap, root_cap)` for hop 1 ($A \to B$ against the root) *and* `is_subset(final_cap, mid_cap)` for hop 2 ($B \to C$ against what B actually held). Same malicious B, same forged `cap_write` delegation, same query — only the verifier's comparison target moves from "root" to "immediate parent."

Pinned output (`analyses/harbor_card_v7_results.txt`):

```
1 RESULT not event(Accepted(cc_2,harbor_id[],cap_write)) is true.
```

The identical escalation attempt v6 accepted is now rejected at the `is_subset(cap_write, cap_read)` step, because write is not a subset of what B was actually holding.

2.9 *(exercise on page 33)* Both models share a grant chain — a first-party caveat folded into the root HMAC, a third-party commitment `vid` keyed by a shared caveat key, and a discharge bound to the request via `hmac(BIND0, ···)`. In v2, two grants (a low-authority one the daemon discharges, and a high-authority one it never discharges) share the same caveat key —

realistic, because one live session's check backs every grant minted under it. v2's naive relay computes `bound = hmac(BIND0, d1)`, binding the discharge to the *caveat* only, never to which grant it is being presented for. An attacker who holds the daemon's genuinely-issued low-authority discharge — public, since it travels in the clear — can present it alongside the public high-authority grant; the relay recomputes the same discharge chain from the shared caveat key and accepts. `RelayAuthorizesNaive` fires for the high-authority grant's signature although the daemon's `DaemonIssuesDischarge` event never named it. This is the macaroon-construction analogue of `analyses/harbor_card_v6_multihop_attack.pv`: a verifier that checks a credential against something coarser than the exact object in front of it — there, the root instead of the immediate parent; here, the caveat key instead of the specific grant — admits a substitution.

v1's verifier folds the grant's own signature into the binding, `bound = hmac(BIND0, (g_sig, d1))`, not just the discharge chain. A discharge minted for the low-authority grant's signature cannot satisfy the high-authority grant's binding check: HMAC is modeled as a one-way keyed PRF with no destructor, so the attacker can replay the low discharge's bytes verbatim but cannot recompute the high-authority binding without the private caveat key only the daemon holds.

Pinned output, v1 (`analyses/macaroon_discharge_v1_results.txt`):

```
RESULT event(RelayAuthorizes(s)) ==> event(DaemonIssuesDischarge(s))
    is true.
```

Pinned output, v2 (`analyses/macaroon_discharge_v2_naive_unsound_results.txt`):

```
RESULT event(RelayAuthorizesNaive(s)) ==> event(DaemonIssuesDischarge
    (s)) is false.
```

Both discharges are unforgeable without the caveat key; the entire gap in v2 is that its binding forgets which grant it was issued for.