dregg

A Verified Distributed Object-Capability Substrate

Ember Arlynx

github.com/emberian/dregg

Abstract

dregg is a distributed object-capability substrate designed so that an absent party can verify a history without re-executing it or trusting its executor. An atomic turn exercises attenuable authority over owned state and leaves a receipt. For histories produced with full-turn proving enabled, recursive aggregation reduces those receipts to one root. Subject to an explicit cryptographic and liveness floor, checking that root establishes that the attested history preserves authorization, value conservation, and faithful state commitment.

The model divides a cell into four substances with different disciplines: linear value, guarded-mutable state, constructively produced authority, and monotone evidence. Holding a capability means being able to exhibit the witness required for an act; delegation may attenuate authority, while new authority must be constructed from connectivity already held and disclosed by a receipt. One predicate algebra expresses caveats, cell programs, turn preconditions, and intent demands, and makes their coordination and disclosure costs explicit.

The Lean 4 kernel defines these transition semantics and is also the executor the node invokes through FFI. The same verified modules emit byte-pinned circuit descriptors. Rust interprets those descriptors but does not author their constraint algebra, and the descriptor prover is the production proving path. The assurance case then connects the running entry to five guarantees — authority, conservation, integrity, freshness, and light-client unfoolability—while pinning each theorem's axiom set and exposing every cryptographic or liveness hypothesis.

Those hypotheses matter. At the deployed domain size the mechanized commit-phase ledger (BCIKS20′s ε_C, fixed-parameter) is a knob-ledger calculation, not an adversarial soundness theorem over supplied proofs, and extraction remains an explicit carrier; the assurance section states the evaluated figure and its caveats. A costed extension-degree-eight configuration exceeds 120 bits under the same ledger but is not deployed. The paper therefore separates the mechanized transition argument, the proof-system floor, and deployment correspondence instead of collapsing them into one claim. The same kernel spans distributed cells and local seL4 protection domains, and factory-minted applications reuse its receipt and theorem boundary rather than introducing a second execution model.

1 Introduction

A distributed object-capability substrate has one hard problem: a party absent from a transition must still be able to verify that it happened correctly. dregg makes the proof witness the protocol's **correct evolution**. Subject to the assumption floor stated in Section 17, a new node, auditor, or phone can check one aggregate root and learn that the attested history was authorized, conserved value, and was faithfully committed. It re-executes nothing and need not trust the executor. The design follows from the adversarial form of this requirement: a server that ran the protocol incorrectly must not be able to convince a light client otherwise.

Each proof-carrying turn is designed along three axes. It is **non-malleable**: every guarantee-relevant field is bound, so the proof cannot be re-pointed at a different state or receipt. It is **precondition-complete**: authority, availability, freshness, and non-amplification are circuit conjuncts, not side checks a prover may omit. And it **refines the protocol**: the attested relation is the kernel's semantics, not a lossy re-encoding. Together these properties exclude the central failure mode — a history that verifies although the kernel could not have produced it — conditional on the proof-system carrier made explicit in Section 4.

1.1 The sentence

The whole system is one sentence, given algebra:

A turn is the exercise of an attenuable, proof-carrying token over owned state, leaving a verifiable receipt.

State lives in **cells**. A **turn** exercises authority — a **token** that may be attenuated on every axis and that carries a proof of its own legitimacy — over **owned state**, and leaves **Q**, a receipt that commits to the result. The section on the model fixes the nouns; the section on authorization gives the authority logic; the section on proofs gives the receipt and its aggregation.

1.2 The organizing asymmetry

dregg treats authority as **constructive knowledge**. To hold a capability is to be able to exhibit a witness that authorizes an act, never merely to assert it. This is the realizability reading of intuitionistic logic made operational, and it rests on one asymmetry:

proof-checking is cheap and trusted; proof-search is undecidable and untrusted.

Every trust decision in the system is a check. Whoever wants to act bears the burden of producing a witness; whoever guards state only ever verifies. A capability is not a key in a lock — it is a proof obligation one can discharge. The asymmetry places search at the untrusted edges (solvers, intent matchers, provers) and checking at the trusted core (the executor, the circuit, the light client). It is the same asymmetry that lets a STARK verifier accept a statement it could not have found.

1.3 What the kernel governs

The kernel governs four **substances**, each under a discipline of use — a law about how it may move through time. Value is **linear**: per asset, the resource sum across a turn is exactly zero.

Authority is **produced under non-forgeability**: it grows only by authorized, receipt-disclosed construction from connectivity already held, and narrows freely along one edge. Evidence is **monotone**: once known, never unknown. State is **guarded-mutable**: it changes only under a predicate, only by its owner. The kernel's signature is seven constructors, with `shield` and `unshield` as the two directions of one evidence operation. Each is assigned a structural role in a substance discipline; the section on the model states the roster and the exact, signature-level independence property proved about that assignment.

The authority discipline is the one most often gotten wrong. A model in which every step only narrows — a monotone descent down a lattice — forbids the patterns that give capabilities their power: a holder introducing a third party, an unsealer combining with a sealed box to yield contents neither names alone, a mint creating fresh resource on an authorized gesture. dregg's authority law is generative **and** disciplined: authority grows, but every generative act is itself authorized by held knowledge and lands on the chain receipt-disclosed. The section on authorization states this as one law, Miller's *only connectivity begets connectivity*.

Everything that constrains a turn — a caveat on delegated power, a program on owned state, a precondition on a turn, a demand on the world — is one algebra of decidable predicates over the proposed step. The predicate the executor evaluates is compiled to the circuit obligation a proof discharges, so an installable guard and a provable property are one mechanism. Two prices are computed rather than assumed. A **coordination dial** classifies whether two concurrent turns merge without coordination: a confluence-stable guard runs coordination-free, one that is not provably so forces ordering, and the classifier prices the difference rather than forbidding the expensive case. A **disclosure dial** governs how much a guard reveals while checking — cleartext, committed, range-proved, jointly-garbled — on the principle that what the proof does not need, it does not ask to see.

1.4 The proofs are about the thing that runs

The semantics are a Lean 4 development that is **also the deployed executor**. The gated whole-forest step `execFullForestG` is compiled, exported through FFI as `dregg_exec_full_forest_auth`, and invoked by the node on its production path. The running-entry guarantee (`AssuranceCase.running_entry_sound`) is stated over exactly this function: conservation, non-amplification, and per-node attestation hold over the entry the node calls, not over an abstract model beside it.

The proof system inherits the same discipline. Each kernel statement carries a descriptor from which both the executor reading and the circuit obligation are obtained, and an agreement theorem welds the two readings (`Circuit.Argus.Receipt.argus_circuit_executor_receipts_agree`). No constraint is authored in Rust, and this is an enforced invariant, not a convention. The hand-written STARK engine is deleted; the descriptor prover (`prove_vm_descriptor2` / `verify_vm_descriptor2` in `circuit/src/descriptor_ir2.rs`), driven by Lean-emitted descriptors, is the only production proving path. The last first-party circuit authored in Rust — the non-revocation freshness check — is retired: its constraints are emitted from `Dregg2/Circuit/Emit/NonRevocationAdjacencyEmit.lean`, and the deployed prover loads the emitted descriptor by name (`circuit/src/dsl/revocation.rs`). Two gates keep this true. A ratcheting test scans every source file in the circuit crates across all three constraint dialects — symbolic builder calls, evaluation closures, and constraint- expression literals — and fails the build on any new or grown Rust-

authored constraint algebra (`circuit-prove/tests/law1_enforcement_gate.rs`). A pull-request check detects re-authored mirrors — a hand-typed copy standing in for a loaded artifact — and runs a canary against itself before each run, refusing to report from a gate that cannot fail (`scripts/check-mirror-gates.sh`).

1.5 The assurance case is an artifact

The system's guarantees are a Lean file, `metatheory/Dregg2/AssuranceCase.lean`. It states five guarantees — authority, conservation, integrity, freshness, unfoolability — plus a running entry that closes the first three over the deployed function. Under each guarantee it assembles the keystone theorems that discharge it, and it `#assert_axioms`-pins every name: the build fails unless each theorem's full axiom set is exactly the kernel triple $\{\text{propext}, \text{Classical.choice}, \text{Quot.sound}\}$. Everything else enters as an explicit floor of eight cryptographic and liveness carriers, stated as hypotheses rather than axioms. On the verified entry there is no trusted-executor premise, no out-of-band "this was authorized" premise, and no guarantee-relevant field of the post-state omitted from the commitment. Section 19 states where host execution and deployment state remain outside that claim.

1.6 Applications inherit theorems

Applications do not extend the kernel; they are cells. A **factory** publishes a descriptor — a slot layout plus predicate constraints — and the `create` verb mints cells from it. From that moment the executor enforces the program on every turn touching the cell. Recurring coordination shapes — escrow, obligations, queues, mailboxes, bridges, sealer/unsealer boxes — ship as verified factories whose safety keystones are kernel theorems, so an application's contract is inherited from the kernel rather than re-established per app. The shape is uniform: value at stake lives in the minted cell's own balance column, so funding and settling are ordinary moves and conservation is the ordinary kernel law with no side tables. The section on realization develops the catalog.

2 The model

This section fixes the nouns — substances, cells, capabilities, assets, turns, receipts — and the verbs. The authorization logic is given in Section 3; the receipt and its aggregation in Section 4.

2.1 The four substances

The kernel governs four substances. Each has a **discipline of use** — a law about how it may move through time — and the kernel is the enforcement of those laws.

substance	discipline	the law
value	linear — moves, never copies or vanishes	per asset, $\Sigma\delta = 0$ exactly, per turn and per attested run
authority	produced under non-forgeability — grows only by authorized, receipt-disclosed construction; narrows freely along one edge	*only connectivity begets connectivity* (Section 3)
evidence	monotone — once known, never unknown	the nullifier and commitment ledgers only grow
state	guarded-mutable — changes only under a predicate, only by its owner	the frame rule

Table 1: The four substances and their disciplines.

The substance algebra is mechanized as a product of resource cameras (`Dregg2/Resource.lean`). A supply camera carries the linear law: a committed transfer is a frame-preserving update of the moved balances (`move_value_fpu`), and a supply-violating mint is not (`mint_not_value_fpu`). An `Auth` camera carries the authority and evidence laws: an amplifying grant is rejected by the camera itself (`amplifying_grant_not_fpu`), confinement of held authority is frame preservation by definition (`ConfinesAuthority`), and erasing a known nullifier is not a valid update (`forget_evidence_not_fpu`). Cells and programs carry the guarded law.

Every kernel verb passes **two gates**. **Admission** is the epistemic half: does the supplied witness realize the demanded predicate? (`Verify P w`; Section 3). **Footprint** is the ontic half: the update is a frame-preserving update of the verb's footprint in the product camera — it respects what the substances **are**. The halves are independent in both directions. The camera is blind to the guards: a write that every caveat rejects, and that the kernel therefore refuses, is still a valid camera update (`camera_blind_to_caveats`). And order-shaped camera validity alone cannot carry exact conservation: over the ledger's integer carrier every authoritative update is frame-preserving (`int_auth_fpu_vacuous`), and even an ordered carrier admits a coordinated mint (`nat_auth_coordinated_mint_fpu`). The value law therefore lives in the issuer supply discipline of Section 2.3, not in the order.

2.2 Cells

A **cell** is the four substances gathered behind one identity:

```
Cell = { operator, lifecycle, nonce,
         registers : Fin R → Value,   -- the small named state-machine fields
         heap_root : sorted-Poseidon2 map over (collection, key) → value,
         bal       : Asset → ℤ,       -- the value column
         clist     : sorted-Merkle of Cap,  -- held authority
         program   : Pred }           -- the guarded-state law, self-imposed
```

Nothing is ownerless; every object in the system **is** a cell or lives in one. Programmable state is a **register file plus a heap**: a small fixed file of named scalar fields — the state machine every turn touches — and one register holding `heap_root`, the root of a sorted-Poseidon2 Merkle map

over `(collection, key) → value` for collections of unbounded size, opened only where a turn reads or writes. This is the same five-domain address space — registers, heap, capabilities, nullifiers, receipt index — that the universal memory model gives one commitment discipline (`Dregg2/Crypto/UniversalMemory.lean`, mirrored by the executor-state projection in `turn/src/umem.rs`). Per turn, the deployed circuit binds the touched cell's register file as flat before/after trace columns and carries its balance limbs, nonce, and commitment roots among the public inputs; the deployed layout of record is the emitted staged registry (`circuit/descriptors/rotation-wide-*.tsv`), and each install appends an operator-stamped row to `docs/VK-REGEN-LOG.md`.

The frame rule — a turn touching one cell leaves every other cell unchanged — is proven once and read four ways: sovereignty (your cell is untouchable), joint turns (disjoint footprints compose as separating conjunction), sharding (disjoint frames commute), and offline operation (a frame advances alone and merges soundly).

A **capability** is the token: target, rights, caveats (a predicate), expiry, revocation epoch. It is attenuable on every axis, revocable by epoch bump, and storable in slots — a capability is a value, and retrieval re-checks the grantor's epoch, so storage cannot launder revocation.

2.3 Assets

An **asset is an issuer cell's promise**: an `AssetId` is the issuer's `CellId`. Mint and burn are the issuer moving value from and to its own well under its own program; fees are ordinary moves to pot-cells whose programs **are** the fee policy. Conservation is therefore one law with no exceptions to disclose: the moved asset's total is exactly invariant (`RecordKernel.recTransferBal_sum_conserve_moved`) and untouched assets are pointwise unchanged (`RecordKernel.recTransferBal_untouched`) — no cross-asset leakage.

2.4 The kernel signature

The kernel signature is eight verbs — seven constructors, with `shield` and `unshield` the two directions of one evidence verb. Each is the structural rule of one substance's discipline. The registry assigns each constructor a (substance, polarity) label and proves that assignment injective (`VerbRegistry.minimality`, `VerbRegistry.each_verb_irreplaceable`). Thus no two constructors occupy the same labeled role. This is a signature-level non-redundancy check; it is not a semantic lower bound proving that no alternative kernel or composition could express the same behavior with fewer primitives.

verb	substance / polarity	structural rule
`create`	birth / introduce	mint a four-substance cell (incl. factory instantiation)
`write`	state / neutral	guarded register/heap/program/permission update under the frame
`move`	value / neutral	exchange of the linear substance (fees and burn are moves to wells)
`grant`	authority / introduce	authorized production / narrowing along one edge
`revoke`	authority / eliminate	epoch-narrowing that stales held authority
`shield` / `unshield`	evidence / introduce	note-create / note-spend: grow the evidence ledger
`lifecycle`	retirement / eliminate	the seal / destroy / sovereign custody automaton

Table 2: The kernel signature. The roster of record is the registry itself (`Dregg2/Substrate/VerbRegistry.lean`), which reifies the wire effect enum one Lean tag per variant.

The wire vocabulary turns actually carry is larger than the signature. The registry classifies it with a total cover (`VerbRegistry.classify`, `VerbRegistry.classify_total`) that sends every reified tag to exactly one of three places: a **kernel verb**; **turn structure** (exercising a capability is a *use*, not a verb; a refusal is an *outcome*; the nonce is prologue; pipelining is composition); or a **factory pattern** — a cell program built from the surviving verbs only. On the live enum the factory bucket is provably empty (`VerbRegistry.no_live_factory_tags`): the families the factories replaced were deleted from the wire, not reclassified. The exhaustiveness check is the Lean compiler's — a tag added to the roster without a classification does not compile. A cross-language cover gate (`turn/tests/verb_registry_cover_gate.rs`) parses the 34-variant Rust `Effect` enum and the Lean `EffectTag` roster as sets and requires exact equality. A new wire variant therefore fails either the Rust–Lean parity gate (no reified tag) or the Lean build (an unclassified tag). The seven post-lockstep additions are classified at their substance boundary: `SetProgram` and `Custom` as guarded state writes, `Mint` as the issuer side of `move`, and `ShieldedTransfer`, `Promise`, `Notify`, and `React` under the evidence discipline, with `React` explicitly pinned to the same classification as a note spend.

2.5 Turns

```
Turn  = auth ∘ body ∘ receipt
body ::= verb | seq | par | hole(Pred)
```

A turn is a forest of actions executed as a transaction: every action admits and every effect lands, or the state is exactly what it was. Each action carries a demand ⊣ supply pair (the cell demands a predicate, the action supplies a witnessed authorization; Section 3), the guards in scope, the proposed effects, and a signed binding to the canonical v3 message — the federation, turn nonce, action, and effects — so a signature cannot be re-pointed to another action body or replayed after the nonce advances. Multi-party turns are the same shape under one commitment, each

participant contributing its own authorization. Conditional and pipelined turns are composition structure, where `hole(Pred)` is a typed hole a counterparty's fulfillment discharges.

A committed turn leaves **Q** — the receipt: the committed postcondition under one commitment scheme (sorted-Poseidon2 Merkle throughout). The witness proves Q; the disclosure dial projects Q; aggregation folds Q; the light client verifies only Q-chains (Section 4). Q is one object in every role.

3 Authority as constructive knowledge

3.1 The thesis

A capability is a piece of constructive knowledge: to hold one is to be able to exhibit a witness that authorizes an act — never merely to assert it. The capability graph is a distributed knowledge graph. Nodes are cells — knowers with private state and a program. Edges are capabilities — directed facts ("this cell can constructively demonstrate authority over that one") carrying attenuated rights. The graph is partial and local: there is no global registry of who-may-do-what; one learns of an edge only when someone presents a witness for it, at the point of use.

The organizing asymmetry of Section 1 is the realizability reading of intuitionistic logic made operational: proof-checking is cheap and trusted; proof-search is undecidable and untrusted. Every trust decision is a check. Whoever wants to act bears the burden of producing the witness; whoever guards state only ever verifies. A capability is not a key in a lock — it is a proof obligation one can discharge.

3.2 The demand ⊣ supply adjunction

Each action is judged by a matched pair: the target cell **demands** `AuthRequired` (a predicate); the action **supplies** a witnessed `Authorization`. Admissibility is `Verify P w` — the counit of the Predicate ⊣ Witness adjunction, whose Galois connection is carried in `Dregg2/Laws.lean`. Guards (the guard algebra) are more predicates over the same step, each either first-party (decidable now) or witnessed (a registered verifier discharges it). The signed binding to the canonical message pins the witness to this exact step, so an authorization proved for one context cannot be replayed into another.

3.3 The production law

The characteristic — and easily-missed — fact: **authority is produced, not merely spent.** A model in which every step only narrows (a monotone descent down a meet-semilattice) forbids exactly the patterns that give capabilities their power. The real dynamics have a generative half and a restrictive half, disciplined by one law.

Generative (the graph grows):

- **Introduction** (Granovetter): a holder of an edge to Carol grants Bob a *new* edge to Carol. Enforced non-amplifying (the conferred edge ≤ the introducer's own), consensual, and time-bounded.

- **Rights amplification**: a held amplifier combines with another fact to yield access neither names alone — the sealer/unsealer pair (unsealer $\otimes$ sealed-box $\vdash$ contents), the brand, the mint. It does not break the discipline precisely because the amplifier is connectivity already held.
- **Powerbox / mint / factory**: a designated authority creates fresh edges or resource on an authorized gesture — the legitimate point where new authority enters.
- **Parenthood / endowment**: creating a cell endows the child.

Restrictive (the graph narrows):

- **Attenuation** narrows the rights on one existing edge (a caveat, a facet subset). The narrow-only rule governs *one edge's rights*; it is not the law of the whole system.
- **Revocation / expiry** removes an edge (epoch-bump; one-way).

The one law: Miller's *only connectivity begets connectivity*. There is no ambient authority; every generative act is itself authorized by held knowledge; and generative and annihilative acts are receipt-disclosed — the conservation typing forces them onto the chain, un-strippably. Authority grows, but only through authorized, **non-forgeable** construction. This is an epistemic non-forgeability invariant, not a lattice descent.

Mechanized: `Spec.only_connectivity_begets_connectivity` (the whole-history closure: every edge in any reachable authority graph either was held initially or descends by conferral from an authorized generative step — none ex nihilo); `EffectsAuthority.introduce_non_amplifying` (a conferred capability is a genuine subset of the held one, over the real attenuation lattice); `EffectsAuthority.amplifying_grant_rejected` (the discriminating direction: a grant conferring authority the holder lacks is rejected — the predicate is two-valued); `AuthModes.captp_granted_le_held` (the dispatcher gate on delivered handoffs); `FullForestAuth.execFullForestG_no_amplify` (every delegation edge of a committed forest, at the running entry).

3.4 Three logics over a step

Each turn is judged by three orthogonal logics.

1. **Conservation — substructural / linear.** Resources cannot be copied or discarded for free; generative and annihilative moves are disclosed exceptions bound into the receipt. Linear logic's structural rules, read as a security law: no inflation, no loss.
2. **Ordering — temporal / modal.** When is a fact final? A finality lattice over one Merkle-CRDT DAG; effects commit at the join of the written cells' tiers and never downgrade. "Knowledge becomes common knowledge" is the modal ascent (the ordering logic).
3. **Independence — the confluence lattice.** Which concurrent inferences commute? The join-semilattice of invariant-preserving merges — the coordination-free fragment, which the guard algebra's coordination dial computes (the guard algebra).

3.5 Crossing a trust boundary

Inside a trust root, authority is **positional**: holding the edge is the proof, and the mediator enforces it. Across a boundary it becomes **epistemic**: one must present a verifiable witness, because the far side shares no mediator. The crossing is a named-lossy functor — *permission*

survives, authority does not — and the loss is load-bearing: confinement and revocable-forwarding are dropped, which is why a forwarded capability is revocable *by construction*. A hosted cell's full state lives with its host; a **sovereign** cell keeps only a commitment and proves its own transitions, so a far federation admits it knowing only how to check a proof, never how to re-run the cell.

4 Proofs: the descriptor circuit and the light client

4.1 The descriptor circuit

The proof system is organized so the circuit is **derived**, not hand-built. No constraint is authored in Rust. Each kernel statement carries a **descriptor** — the structured form of its semantics — from which the executor reading (`interp`) and the circuit reading (`compile`) are both obtained, with agreement theorems welding them: `Argus.Receipt.argus_circuit_executor_receipts_agree` is the receipt-level weld, and the per-effect statements live in `Circuit/Argus/Effects/`. One term has two provably-agreeing readings: the turn is a proof term, the circuit is the logic's proof checker, a receipt is a judgment, and the chain is one growing proof object.

One proving path exists. The earlier hand-written STARK engine is deleted from the tree; `prove_vm_descriptor2` and `verify_vm_descriptor2` (`circuit/src/descriptor_ir2.rs`) are the production prove and verify entry points, and every consumer runs them over the Lean-emitted, byte-pinned descriptor artifacts. A coverage gap is closed by emitting from a proved Lean module, never by authoring a constraint by hand.

The proving stack is a STARK over Plonky3 (BabyBear field, FRI) [1], [2], with the commitment scheme of Section 4.3 inside the arithmetization (Poseidon2) [3] and recursion folding receipts into the aggregate the light client checks. The circuit layer adds exactly one assumption to the floor of the assurance case — the named engine-soundness carrier `EngineSound.recursive_sound` — and no other.

4.2 The engine carrier, quantified

The one recursion assumption has a measured strength, and the measurement reads differently on its two sides.

The recursion apex configured for the light-client proof runs at log-blowup 6 with 19 queries and 16 bits of query grinding, over tables floored at height 2^{16}, so its initial evaluation domain has 2^{22} points. Under the batched-FRI soundness theorem of Ben-Sasson, Carmon, Ishai, Kopparty, and Saraf (BCIKS20, the bound the deployed prover's source cites), composed by the ethSTARK minimum rule, the parameter ledger evaluates to 57.98 bits. At the apex the commit-phase term binds, so adding queries or grinding does not improve this particular calculation; only the evaluation-domain term moves it. Under the successor proximity-gaps bound (BCSS25) the same parameters evaluate to about 71 bits, but no published theorem composes that bound into a FRI soundness statement. Neither number is presented here as an adversarial soundness level.

What 57.98 is: an arithmetic evaluation of the published bound at the deployed parameters, validated against the source papers' own worked examples. The mechanized part of the ledger is a density statement over a supplied proof: the set of folding challenges that let a far word survive a fold has bounded cardinality, and its density in the challenge field is below 2^{-b} for the ledger's reported b (`FriLedgerSound.ledger_perFold_error`). The ledger's per-fold and query columns are provably independent, so no single number summarizes a configuration (`FriLedgerSound.query_ledger_does_not_determine_perFold`). What 57.98 is not: a bound on an adversary. The metatheory's verifier is a Boolean function of a supplied proof (`FriVerifier.verifyAlgo`); no adversary model and no grinding model appear in its statement, and the extraction hypothesis — everything FRI is trusted to deliver on an accepting run, `AlgoStarkSoundTransferV3.FriLdtExtractV3` — enters the circuit-soundness reduction as a hypothesis, not a theorem. The calculation characterizes one necessary component of the carrier; it does not discharge the carrier or price proof forgery against an adversary.

A configuration that reaches a conventional target is identified and priced. At extension degree 8 over BabyBear, with log-blowup 6, 36 queries, and 16 grinding bits, the same ledger reads 122.60 bits on every shipped configuration, leaf and apex alike, with 2.6 bits of margin over 120. Cutting over costs a rewrite of the extension-field arithmetic in the BN254 wrap (hand-unrolled at degree 4 across roughly 136 sites), a verifying-key re-key across every consumer, a measured 25% more prover time, and about 20% more proof size. The deployed configuration is degree 4; the cutover is priced, not taken.

4.3 Q

A committed turn leaves **Q** — the receipt: the committed postcondition of the step under one commitment scheme. State, capability lists, and aggregates alike commit under a sorted-Poseidon2 Merkle structure. One object serves every role: the witness proves Q, the disclosure dial projects Q, aggregation folds Q, the light client verifies only Q-chains.

Two coupled chains record the epistemic history. The **receipt chain**, per cell (prev $\rightarrow$ hash), is the append-only log of what was inferred — the log is the truth, the database is a cache. The **witness bundle**, bound into each receipt by its witness hash, is the content that makes the inference replayable. Checkpoint, restore, replay, and time-travel are theorems, not features: reseed the execution from a recorded witness and the same receipts fall out.

4.4 The receipt binds the whole post-state

The commitment is determined by — and, under collision-resistance, recovers — every field of the post-state, including the frame the step did not touch. Mechanized:

- `Argus.Receipt.argus_commits_to_one_receipt` — the circuit term commits to exactly one receipt, determined by the post-state;
- `Argus.Receipt.argus_circuit_executor_receipts_agree` — the circuit's receipt and the executor's receipt are the same judgment (one semantics, two readings);
- `CommitmentCrossBind.runnable_binds_same_system_roots` — equal commitment roots imply equal full state (cells *and* rest-of-state);
- `CommitmentCrossBind.chC_bad_not_bridge` — the negative direction: a commitment that drops a field is rejected as a bridge, so the binding is not vacuous.

Freshness is part of the same object: a committed spend's nullifier was provably fresh and a repeat fails closed *at the term level* (`noteSpendStmt_no_double_spend`, `noteSpendStmt_replay_rejected`), with non-membership itself a witnessed claim — a sorted-tree non-membership opening with a circuit gate (`NonMembership.nonmembership_sound` / `NonMembership.nonmembership_complete`). Freshness is checked by an opening, never by a Merkle-path scan of the whole nullifier set.

4.5 Aggregation

Proofs fold. Each step's proof attests its own statement plus the proof before it; the seam is pinned — step i's post-root must be step $i+1$'s pre-root (`HistoryAggregation.wellformed_attests_whole_history`, `HistoryAggregation.root_tooth_pins_state`). Aggregating witness hashes into one root is incremental verifiable computation [4]: the DAG of all knowledge compressed to a single checkable claim.

Proofs are **additive attestation, not permission**. Between parties that trust each other's execution, turns flow as signed receipts and nobody waits on a prover. The proof layer exists so that someone who was not there — a new node, an auditor, a phone — can verify the whole past at the cost of one check.

4.6 The light-client theorem

The keystone is `RecursiveAggregation.light_client_verifies_whole_history`. A verifier that checks only `verify agg.root` — re-witnessing nothing — learns that every turn in the history executed correctly, in order, and that the final root is a genuine fold of that history. The composed guarantees of the assurance case ride along: the whole attested history conserves (`RecursiveAggregation.attested_history_conserves`), and tampering is not merely detectable but **unprovable** — a reordered chain forces the binding predicate false, so no verifying aggregate exists (`RecursiveAggregation.tampered_aggregate_cannot_bind` `RecursiveAggregation.leaf_pairing_defeats_swap` rules out re-pointing a verifying leaf at a different step).

The realization on the executable term IR is the Argus strand (`Argus.Aggregate.argus_strand_light_client`, `Argus.Aggregate.tampered_argus_strand_rejected`), and the one recursion obligation is the named engine-soundness carrier `EngineSound.recursive_sound` on the assumption floor of the assurance case, quantified above. This is unfoolability made precise: a light client cannot be convinced of a history the protocol did not actually produce.

5 The guard algebra

Everything that constrains a turn is one algebra of decidable predicates over the proposed step. The guard the executor evaluates and the obligation a proof discharges are compiled from the same object: an installable guard and a provable property are one mechanism. This section gives the algebra, its closures, its compiled form, and the two prices it computes.

5.1 One guard, four polarities

A guard is one object (`Spec.Guard`):

```
Guard = firstParty(p) ⊕ all/any/not ⊕ witnessed(s)
```

A **first-party** branch is decidable now, against the old and new state. A **witnessed** branch defers its statement to the verify seam (`Spec.Guard.admits`). That seam is the single site where external evidence enters the grammar; a third-party discharge, a range proof, and an arbitrary proof-carrying claim all arrive there. The novelty is not the grammar but its reach: the one algebra appears at four **polarities** that are four separate mechanisms in most systems.

polarity	**the predicate is imposed on**
caveat	*delegated* power — a restriction travelling with a capability
program constraint	*owned* state — the cell's self-imposed law
precondition	a *turn* — what an action requires to be admissible
intent demand	the *world* — the typed hole `hole(Pred)` a counterparty's fulfillment discharges (Section 2)

Table 3: The four polarities of one predicate algebra. A caveat on a macaroon, a smart-contract invariant, a transaction guard, and an open-order condition are one mechanism here.

Because the four are one object, a caveat is checkable by the proof system, not only by the issuing service; an intent demand is the same kind of fact as a cell program; and a precondition compiles to a circuit obligation exactly as an invariant does. An intent is the demand polarity made first-class: a fulfillment supplies the witness that closes the hole, and the counit of Section 3′s demand $\dashv$ supply adjunction is what discharges it.

5.2 The grammar and its closures

The first-party atoms are small decidable shapes over a cell's slots — equality, bounds, write-once, immutability, monotone updates, deltas, sums, allowed-transition automata, membership, prefix, and typed identity atoms over symbol and digest slots. The catalog of record is the `Exec.StateConstraint` inductive; the executor's Boolean algebra over it (`PredAlgebra.Pred`, evaluated by `PredAlgebra.Pred.eval`) closes the atoms under negation, conjunction, and n-ary disjunction at every level. Two further closures keep the grammar from ossifying into axis-aligned special cases.

Relational closure (`Authority/RelationalClosure.lean`). Any affine relation over the post-state — $\text{head} \le \text{tail} + \text{capacity}$, $\Sigma\,\text{slots} = \text{const}$ — is the same predicate object (`RelationalClosure.RelPred`), with `RelationalClosure.ofFieldLteOther_eq` recovering the single-slot atoms as instances. There is no new atom per shape: the guard language is the internal predicate logic of the state object, bounded only by decidability and circuit-expressibility.

Quantified closure (`Authority/QuantifiedPredicate.lean`). Bounded $\forall$ and $\exists$ over slot ranges compile to the relational closure with a proven constraint budget: `QuantifiedPredicate.compile_sound` welds the compiled form to the quantified meaning, and

`QuantifiedPredicate.andFold_budget` / `QuantifiedPredicate.orFold_budget` bound the cost. Quantifiers cost what they cost, and the cost is a theorem.

5.3 The compiled form

The comparison atoms compile to circuit descriptors authored in Lean, not to circuits authored beside the kernel. The threshold descriptor (`PredicatesArithmeticEmit.predicateGeDesc`, registered as `dregg-predicate-arith-ge::threshold-v1`) is representative. It proves a conjunction with a shared variable: the compared value satisfies value $\geq$ threshold for a public threshold, **and** the public fact commitment hashes over the same column being compared, binding the claim to committed token state. Without the shared column the circuit would prove an inequality about a number the prover chose. The comparison refuses by range: the circuit range-proves diff = value $-$ threshold into $[0, 2^{29})$, and a value $<$ threshold witness wraps `diff` far outside that interval in the field, where no limb decomposition exists.

The descriptor bytes are pinned twice. The Lean source pins its own emitted wire string with a `#guard`, and an emit gate (`circuit-prove/tests/predicates_arithmetic_emit_gate.rs`) embeds the identical string, proves an honest witness through the node's one proving entry (Section 4), and runs mutation canaries that each tamper one thing and assert the refusal bites the named constraint. Sibling descriptors cover the $<$, $\leq$, $>$, $\neq$, and in-range atoms, the relational and compound forms, and the temporal predicate, each with its own gate. Descriptor installs are recorded in the append-only regeneration log (`docs/VK-REGEN-LOG.md`); the predicate-arithmetic family's current registry entry is the 2026-07-16 row.

5.4 The coordination dial

Guards carry a coordination price, and the system computes it (`Authority/ConfluenceClassifier.lean`) rather than assuming it. Two concurrent turns merge without coordination exactly when their guards are **confluence-stable** — the invariant survives merging independently-taken steps. The classifier is the independence logic of Section 3 landing in the guard language:

- `ConfluenceClassifier.keeps_iff_coordinationFree` — a guard preserves confluence if and only if it runs coordination-free;
- `ConfluenceClassifier.monotone_keeps` — monotone thresholds are free;
- `ConfluenceClassifier.bounded_breaks` / `ConfluenceClassifier.bounded_forces_ordering` — an upper bound forces ordering (consensus).

The classifier does not forbid the expensive case; it **prices** it. A confluence-stable guard runs coordination-free; a guard that is not provably so forces ordering, and the difference is reported, not legislated.

5.5 The disclosure dial

The second computed price is disclosure: how much a guard reveals while being checked. The principle is that what the proof does not need, it does not ask to see. The ladder, most to least disclosed:

rung	what the guard sees
cleartext	the predicate reads values directly
committed	values behind Pedersen commitments; conservation checks homomorphically without opening
range-proved	only the bound is disclosed
jointly garbled	two parties evaluate a shared gate over private inputs and learn the verdict and nothing else

Table 4: The disclosure dial. Each rung is an evaluation mode of the **same** predicate; the law is rung-invariant.

The ladder is mechanized so that privacy never changes the law it checks. Committed evaluation checks conservation homomorphically (`PrivatePredicate.private_conservation_checks_homomorphically`), and the committed and cleartext judgments provably agree (`Spec.committed_iff_cleartext`) — moving down the dial hides inputs, not verdicts. A range proof discloses only the bound (`RangeProof.disclosure_only_the_bound`, `RangeProof.committed_inequality_via_range`). The garbled rung lets two parties evaluate one gate over private inputs (`GarbledJoint.garbled_input_private`, `GarbledJoint.joint_turn_private_gate`), and its disclosure floor is acceptance-only — the verdict and nothing else (`GarbledJoint.garbledDialFloor_is_bot`). This rung's proof path is the same descriptor prover as every other: the garbled-evaluation descriptor is authored in Lean (`GarbledEvalEmit.garbledEvalDesc`), byte-pinned there, and mutation-gated in `circuit-prove/tests/garbled_eval_emit_gate.rs`. The descriptor proves the decryption algebra over the garbled tables; the Poseidon2 garbling-hash binding is a named carrier computed by the executor, and Section A gives the construction and its scope.

Selective disclosure of a receipt — hide, reveal, predicate, committed-threshold — is the same dial applied to **Q** (Section 4): a projection of the receipt, not a second copy of it. Disclosure and proof are one object viewed at a chosen resolution.

6 Ordering and finality

The conservation logic of Section 3 asks whether a step is allowed; the ordering logic asks **when a fact is final**. It is the modal half of the step logic: a finality lattice over a Merkle-CRDT DAG, gated on one named liveness carrier. Safety is unconditional modulo the cryptographic floor; only liveness rests on the carrier.

6.1 The lace

Participants do not share a single linear log. Each keeps a **strand** — an append-only, signed log only its owner can extend — and the strands together form a cross-feed causal DAG, the **blocklace** [5]. A block is signed, content-addressed, and carries a monotone per-creator sequence; equivocation is a structurally detectable incomparable pair, and an honest finalization is unforkable (`StrandIntegrity.forked_strand_not_forkFree`, `Consensus.honest_finalization_unforkable`). Two replicas that merge the same causally-closed blocks reach the same lace and therefore the

same executed state (`LaceMerge.merge_convergence_to_state`) — eventual consistency upgraded to verified convergence.

6.2 Finality as a lattice

Finality is not a boolean but a lattice of tiers over the DAG. An effect commits at the join of the written cells' tiers and never downgrades; "a fact becomes common knowledge" is the modal ascent up that lattice. The finalization rule is deterministic and yields a single anchor per wave (`BlocklaceFinality.finalLeaders_one_per_wave`, `BlocklaceFinality.tauOrder_deterministic`), so two honest verifiers that see the same finalized blocks compute the same order. An equivocator is repelled from approval (`Consensus.equivocator_repelled_from_approval`), and a reconfiguration cannot rewrite already-finalized state (`Consensus.no_conflicting_finalized_state_reconfig`).

6.3 Safety below the threshold; liveness above synchrony

The resilience analysis separates the safety threshold t_S from the liveness threshold t_L, with $t_L < t_S$. Safety holds below t_S (`Consensus.safety_holds_below_tS`), and the bound is real rather than decorative: a negative theorem exhibits a safety break above it (`Consensus.safety_can_break_above_tS`). Liveness is reduced to a single named assumption rather than asserted: under partial synchrony with an honest supermajority, post-GST progress holds (`Consensus.leaderless_progress` from `Consensus.PostGSTProgress`). This is the one liveness carrier on the assurance floor (Section 17); the proofs do not diffuse it through the safety arguments.

6.4 The recorded federation experiment

The implementation has been exercised as a four-validator federation: distinct Ed25519+ML-DSA validator keys under one committee genesis, blocklace synchronization over QUIC, and finality gated on committee quorum rather than a single node (`docs/LOCAL-FEDERATION.md`). In the recorded two-machine run, a turn submitted to one node was super-ratified and its receipt and finalized height replicated identically across all four nodes (`docs/STAGE5-N4-RESULT.md`). The experiment also exposed the committee-size tradeoff. At $n = 3$ the threshold is unanimity, so one asymmetrically delivered block can stop a wave; at $n = 4$ a wave-closing round tolerates one laggard. With the verified ordering gate authoritative, Linux release and Darwin debug builds of the Lean executor committed byte-identical roots for the same turns. The run was an empirical validation, not a claim of a continuously operated federation. Its remaining observed cost was performance: recomputing verified order over the whole lace on each poll can make finality lag block production during catch-up churn, while the DAG and committed state remain consistent (`docs/CROSS-MACHINE-FINALITY-FINDING.md`).

6.5 Revocation is consensus-bound

The ordering logic is where revocation earns its meaning. Bumping a revocation epoch is a state change like any other; it **takes effect** exactly when the relevant views agree the epoch advanced, which is a finality fact, not a local one. Revocation therefore needs consensus (`Liveness.revocation_needs_consensus`), and its dual — whether a capability is dead from a

given partial view — is undecidable (`Liveness.dead_undecidable`), pinned alongside to state the limit. A partitioned network stalls a revocation's finality; it cannot forge the revocation, and it cannot un-revoke. Safety survives partition; liveness is exactly as strong as the carrier.

6.6 Sovereignty across the fabric

A hosted cell's full state lives with its host federation; a **sovereign** cell publishes only a commitment and proves its own transitions, so a far federation admits its turns knowing only how to check a proof, never how to re-run the cell. Two sovereign cells can exchange signed transitions directly and reconcile on reconnect — the same `Pred` and the same receipt algebra, off the consensus path until one party publishes. The ordering logic bounds where coordination is **required**; Section 5′s coordination dial is what tells a guard whether it needs the lattice at all.

7 The realization

The model of the previous sections is not a description **of** the running system; it **is** the running system. This section is the realization: the Lean kernel as the deployed executor, the trust base around it, the single proving path, the factory userspace and two of its customers, what runs today, and the client surface.

7.1 The Lean kernel is the executor

The gated whole-forest step `FullForestAuth.execFullForestG` (`Dregg2/Exec/FullForestAuth.lean`) is compiled, exported through FFI as `dregg_exec_full_forest_auth` (`Dregg2/Exec/FFI.lean`), and invoked by the node on its production path. The running-entry guarantee (Section 17) is stated over exactly this function: `AssuranceCase.running_entry_sound` proves conservation, non-amplification, and per-node attestation of the entry the node calls, not of an abstract twin.

The gate is Section 2′s two-gate discipline made operational. Admission — credential validity, capability authority, and caveats discharged, the `FullForestAuth.gateOK` conjunction — is evaluated fail-closed. Any failing leg rejects the entire forest (`FullForestAuth.execFullForestG_unauthorized_fails`). The substance laws ride through the gate unchanged: conservation and non-amplification (`FullForestAuth.execFullForestG_no_amplify`) are proved over the gated entry, not merely over the raw step.

7.2 The trust base

Around the kernel is a named, surveyed set of host components: the FFI marshalling, the node's admission gates, the standalone and succinct verifiers, the canonical codecs, the bearer-token cryptography, and consensus safety. Transport, storage, and networking sit **outside** the boundary — commitments catch tampering below them, so they must be **available**, not **correct**. The node reports which semantics produced each committed state, live, at `GET /api/node/producer` (`node/src/api.rs`); Section 19 discusses the host-context seam this endpoint surfaces.

7.3 One proving path

The circuit is **derived**, not hand-built (the descriptor reading of Section 4), and the repository enforces this as an invariant rather than stating it as a policy. There is one proving path. The earlier hand-written STARK engine is deleted, and every production proof goes through `prove_vm_descriptor2` / `verify_vm_descriptor2` over descriptors emitted from Lean. Every deployed first-party circuit is emitted from a proved module. The last hand-authored one, the revocation-freshness circuit, is now the emitted descriptor `dregg-non-revocation-adjacency::poseidon2-fact-v1` (`Dregg2/Circuit/Emit/NonRevocationAdjacencyEmit.lean`), byte-identical between the emitter's output and the deployed artifact. A ratchet keeps the invariant from decaying: a gate test (`circuit-prove/tests/law1_enforcement_gate.rs`) counts constraint sites in every circuit source file across all three Rust constraint dialects — symbolic builder calls, evaluation closures, and constraint-expression literals — and fails the build if any file grows or a new one appears; counts may only shrink. The sites that remain are interpreters of Lean-authored constraints, proved-faithful lowerings, and drift detectors, each listed in the gate with its reason.

The two readings of a descriptor — the executor's `interp` and the circuit's `compile` — are welded by the receipt-level agreement theorem (`Argus.Receipt.argus_circuit_executor_receipts_agree`), with the per-effect statements in `Circuit/Argus/Effects/`. The proving stack is a STARK over Plonky3 (BabyBear, FRI) [1], [2], with the Section 4 commitment scheme (Poseidon2) [3] inside the arithmetization and recursion folding receipts into the aggregate the light client checks. The circuit layer adds exactly one assumption to the floor, the named engine-soundness carrier `EngineSound.recursive_sound` the assurance case (Section 17) states what that carrier is currently worth in quantified terms.

7.4 The factory userspace

Applications do not extend the kernel; they are cells. A **factory** publishes a descriptor — a slot layout plus `Pred` constraints — and the `create` verb mints cells from it; from that moment the executor enforces the program on every turn touching the cell. The recurring coordination shapes ship as verified factories whose safety keystones are kernel theorems:

factory	**kernel-theorem safety keystones**
escrow (`Apps/EscrowFactory.lean`)	conditional settlement: `EscrowFactory.no_double_resolve`, `EscrowFactory.release_requires_condition`, `EscrowFactory.release_conserves` / `EscrowFactory.refund_conserves`, and the not-stranded pair `EscrowFactory.open_releasable` / `EscrowFactory.open_refundable`
obligation (`ObligationFactory.lean`)	bonded proof obligations
bridge (`BridgeCell.lean`)	lock / finalize-to-pot / cancel
queues, inboxes, pubsub	value- and capability-bearing mailboxes as bounded state machines
caps-in-slots	sealer/unsealer boxes, sturdy references, handoff certificates — a stored capability is a value, and retrieval re-checks the grantor's revocation epoch

Table 5: The factory userspace. Each pattern is a cell program built from the surviving verbs; its contract is a kernel theorem.

The shape is uniform. Value at stake lives in the minted cell's own balance column, so funding and settling are ordinary `move`s and conservation is the ordinary kernel law with no side tables; the lifecycle is a slot governed by a `Pred` state machine. The runtime mirrors the Lean: `cell/src/blueprint.rs` builds per-deal descriptors whose constraints **are** the verified state machines, and `sdk/src/factories.rs` emits the corresponding turns from surviving verbs only. Applications **inherit theorems**: the Verify toolkit (`Dregg2/Verify/{Contract,Frames,Tactics}`) lets an application state its contract and discharge it by consuming receipts against descriptors, so the kernel's guarantees flow upward without enlarging the kernel.

A delivered cross-session handoff is admitted only through a verified non-amplification gate (`AuthModes.captp_granted_le_held`): the CapTP session surface — sturdy refs, three-party handoff, promise pipelining — is kept out of the consensus-visible kernel, where pipelining is turn composition and a reference is a capability in a slot.

7.5 Games: inherited theorems, worked

The game portfolio is the factory pattern's first customer, and a game turn is the paper's opening sentence taken literally: the exercise of an attenuable, proof-carrying token over owned state, leaving a verifiable receipt. Three games run on this shape (`docs/GAME-STRATEGY.md`): a daily dungeon crawl whose lethality, permadeath, and progression rules are cell programs on the executor path — an attested narrator proposes, the verified rules dispose — and two board games whose rulebooks are Lean definitions.

For automatafl, the staged circuit's admission relation is exactly the graph of the rulebook's turn function (`Games.Automatafl.airAutomatafl_iff_applyTurn`), and a successor board that relocates a piece anywhere the rules do not has no satisfying witness (`Games.Automatafl.airAutomatafl_forged_refused`). For the hidden-hand tug game, a play is admitted exactly when it is legal, the played cards open against the committed hand, and

the successor is the rulebook's (`Games.MultiwayTug.airPlay_iff_applyAction`). The Rust game crates are tested against these statements rather than beside them: a refinement battery drives the built circuit against a reference oracle mirroring the Lean rules, rejecting wrong successors, invalid moves, and forged conflict resolutions (`dregg-automatafl/tests/refinement.rs`); and the automaton-step circuit proves as a recursion-foldable leaf whose in-circuit commitment matches the host binding, folds into a turn chain, and passes the light client's `verify_history` (`dregg-automatafl/tests/prove_fold.rs`). The consequence is the application-level restatement of unfoolability: the operator cannot misreport a hit-point total, and a completed run is checkable by a stranger from its receipts alone.

7.6 Governance as a factory customer

Governance is the same pattern at civic scale, and it is realized (`dregg-governance/`). One executor-backed vote engine drives federation self-governance, community polls, and collective choice over shared state. A ballot is a write-once slot; a tally is a monotone field; the committee quorum rule (two-thirds plus one) is an in-cell affine gate, and resolving a decision must additionally exhibit the required number of distinct approvers through a counting gate. Enactment fires only when the executor's decision-turn commits and the constitution manager independently agrees. The threshold rule is therefore enforced by the same executor as any other cell program, and a governance outcome carries the same receipt as a transfer.

7.7 Executed artifacts and recorded deployments

The verified executor commits turns on the node path, and each node reports per-effect producer coverage at `GET /api/node/producer`. A recorded four-node, two-machine experiment stream-finalized attested turns and committed byte-identical state across operating systems and build profiles (`docs/STAGE5-N4-RESULT.md`); it is evidence about the implementation, not a claim that a durable public federation is presently operated. The game surface has been served from one host through the units in `deploy/games/`, with replay verification and a currently non-durable receipt-anchoring node (Section 13). Every descriptor install appends an operator-stamped row to the regeneration log (`docs/VK-REGEN-LOG.md`), whose tamper evidence is git history.

7.8 The client surface

The client side is the **cipherclerk**: key custody, attenuable tokens, delegation and sub-agent derivation, and the selective-disclosure dial (hide / reveal / predicate / committed-threshold) as literal **Q**-projections (Section 5). The SDK (`sdk/`) is client-local — turn-building, attenuation, and proof generation run on the user's device, and witness data stays there. Search lives at this edge, where Section 1 places it: solvers, intent matchers, and provers produce witnesses; the kernel only ever checks them.

7.9 One model, projected

This userspace service is one realization of the model, not its boundary. The remaining sections develop the substrate it deploys onto: the proof architecture that lets the circuit's shape rotate under proof (Section 8), the capability carried across the distance parameter (Section 9) and its

three faces (Section 10, Section 11, Section 12), and the assurance case that pins every guarantee (Section 17).

8 The proof architecture: ARGUS and path preservation

Section 4 states what a proof says: a verifier holding one root learns the whole history. This section is how that claim survives change in the proof system itself. The arithmetization is not frozen — the field, the commitment layout, and the per-effect circuits evolve — and a verified substrate has to guarantee that evolution never opens a gap the Section 1 adversary, the server that ran the protocol wrong, can slip through. The discipline that guarantees it has three parts: the circuit **witnesses correct evolution** (ARGUS), the circuit shape **rotates** under proof, and every finalized turn **stays proven across shapes**.

8.1 ARGUS: the circuit witnesses correct evolution

The design principle of the proof layer is that the proof is not an external certificate **about** a run but an internal witness **of the protocol's correct evolution**. A light client that checks one aggregate root is fooled exactly when it can be convinced of a transition the kernel would not have produced; the circuit is built so that no such proof exists. This is the negation Section 1 names — a history that verifies but never happened — and the circuit architecture is derived from ruling it out, not from matching whatever the executor's host implementation happened to compute.

Unfoolability decomposes into three properties the per-turn proof must have jointly, and each is a discharged obligation:

- **non-malleable** — the proof binds every guarantee-relevant field of the transition, so it cannot be re-pointed at a different pre-state, post-state, receipt log, or system root. The state-commitment binding below is exactly this property made a fact of the arithmetized field (`RotationLayout.rotatedCommit_binds`).
- **omits no precondition** — authority, availability (freshness against the nullifier set), freshness, and non-amplification are **in-circuit conjuncts**, not side conditions the prover may quietly skip. A verifying proof exists only if every one of them held; an amplifying grant or a replayed spend forces the binding predicate false (`EffectsAuthority.amplifying_grant_rejected`, `noteSpendStmt_replay_rejected`). A forgotten precondition is the classic way a verified system is fooled, so the circuit makes forgetting one unrepresentable.
- **refines the protocol** — the property the proof attests is the kernel's own semantics, derived from the unfoolability requirement, never a lossy re-encoding a host implementation happened to produce. The two readings of one descriptor (below) are what make this a theorem rather than a discipline.

Concretely, ARGUS is the two-readings discipline of Section 4 carried to every effect. Each kernel statement is a descriptor; the executor reading (`interp`) and the circuit reading (`compile`) are obtained from the **same** descriptor, and the receipt-level weld `Argus.Receipt.argus_circuit_executor_receipts_agree` proves they cannot disagree. Because the circuit reading is generated, not hand-authored, there is no

second source of truth to drift: a coverage gap is closed by emitting from a proved Lean module. The per-effect statements live in `Circuit/Argus/Effects/`; the aggregate realization is the Argus strand (`Argus.Aggregate.argus_strand_light_client`, `Argus.Aggregate.tampered_argus_strand_rejected`).

8.2 The state commitment binds the whole post-state

The fidelity floor is the per-turn state commitment: the field from which the proof's pre- and post-state are read. It is a chained Poseidon2 sponge over a fixed limb layout — the cells root, the application registers, the capability / nullifier / heap map roots, and the lifecycle, epoch, and height fields — so that, under permutation collision-resistance, equal commitments imply equal state in **every** limb, including the frame the step did not touch (`RotationLayout.rotatedCommit_binds`). The binding is non-vacuous limb by limb: tampering with the heap root (`RotationLayout.rotatedCommit_binds_heapRoot`), a named register (`RotationLayout.rotatedCommit_binds_named_field`), or the receipt log — by omission, reorder, extension, or truncation (`RotationLayout.rotatedCommit_binds_log`, composing `MMR.mroot_injective`) — produces a distinct commitment. The chained absorption itself is the circuit-side construction `wireCommit`, proved collision-resistant against the same floor (`EffectVmEmitRotation.wireCommit_binds`). This is the Section 4 "receipt binds the whole post-state" made a property of the arithmetized field, so a proof cannot certify a post-state that differs anywhere from the one the executor produced.

8.3 Rotation: the circuit shape evolves under proof

The commitment layout and the per-effect circuit cohort are not fixed constants; they **rotate**. A rotation is a coordinated regeneration of the arithmetization — the limb layout, the per-effect descriptors, and the verifying key together — carried out under a single discipline: the rotated shape is proven to enforce the same semantics as the shape it replaces **before** any verifying key changes. The mechanism is one parametric transformation, `EffectVmEmitRotationV3.rotateV3`, that appends the rotated commitment block to any per-effect descriptor, and two keystones that make the rotation safe to land:

- **equivalent enforcement** — a rotated descriptor forces exactly the pre-rotation per-effect satisfaction semantics, so rotating adds no new per-effect proof obligation (`EffectVmEmitRotationV3.rotateV3_satisfiedVm_v1`);
- **equal published state** — a pre-rotation and a rotated witness of the same effect publish the same state (cells root, registers, map roots, nullifiers), so the rotation cannot change what a turn means (`EffectVmEmitRotationV3.rotV3_binds_published`).

Rotation is a **staged-additive-then-cutover** operation: the rotated cohort is generated and proven beside the live path, with the legacy path byte-identical and its drift guards green, and the verifying-key change is a single deliberate step that makes the rotated shape live. The economics are measured before that step, not assumed: a real multi-operation heap turn proves at a measured size, and the always-paid register limbs versus the metered heap limbs are priced from that measurement. The cell-side and circuit-side state shapes are kept identical by a differential that takes a real turn's post-state through both the cell's commitment and the circuit's trace and asserts they agree, with anti-tamper teeth.

8.4 Path preservation: every finalized turn stays proven

Rotation raises a sharper obligation than "the new shape is sound": **every turn the system has ever finalized must remain provable on the live path, including turns whose shape is heterogeneous.** A single turn may exercise several distinct effect kinds, and an actor's cell may carry arbitrary fields and capabilities — shapes a single monolithic per-effect leg does not cover. Path preservation is the composition that covers them without authoring a new circuit: a heterogeneous turn is split into maximal **cohort-runs** — contiguous spans whose effects all resolve to one rotated descriptor — each cohort-run proves through its Lean-emitted descriptor, and the legs are **chained** by an adjacency check, each run's post-commitment equal to the next run's pre-commitment. A homogeneous turn is one run, byte-identical to the single-leg case; the chain only generalizes it.

The composition is verifier-side arithmetic over the existing Lean-emitted descriptors — it authors no new constraint (the Section 4 law). This is the load-bearing soundness point, so it is worth stating exactly why it is clean: each cohort-run is homogeneous and therefore proves through one of the rotated cohort descriptors already emitted from Lean, whose per-leg binding pins that leg's pre- and post-state (`EffectVmEmitRotationV3.rotV3_binds_published`); the split is a pure fold over the descriptor resolver; and the only new verifier obligation is **equality of two already-proven public inputs** — one leg's post-commitment against the next leg's pre-commitment — a chain break being a typed rejection. The verifier also pins each leg's effect span by re-deriving the cohort split from the turn's effects (so a prover cannot substitute a different-but-chaining effect multiset) and sums the per-leg net deltas to the turn's declared total, so conservation rides the chain. Comparing independently proven public inputs introduces no new statement about what a transition **means**, which is why the composition needs Lean only for review — confirming each per-cohort theorem binds first-row-pre and last-row-post, which a multi-row run already satisfies — and not a new emitted circuit.

The result is the Section 4 light-client guarantee made **robust to the proof system's own evolution**: the aggregate a light client checks (`RecursiveAggregation.light_client_verifies_whole_history`) folds the rotated, chained legs, so no finalized turn need fall back to an unverified path on account of its shape.

Scope. The rotated cohort covers every live effect kind. The resolver that names a descriptor for an effect returns one for every selector the wire enum carries; its fail-closed arm is reached only by the structural no-op and by unknown selectors, and a registry test pins the resolver's output to the exact membership of the Lean-emitted registry (`residue_is_empty_every_live_selector_resolves` in `circuit/src/effect_vm/trace_rotated.rs`). The two effect kinds that formerly resolved outside the cohort now resolve inside it. Capability revocation proves through its rotated descriptor, a held-membership map read composed with a zero-value removal write; separately, the revocation-freshness circuit — the last deployed first-party circuit whose constraints were authored in Rust — is emitted from a proved Lean module (`Circuit/Emit/NonRevocationAdjacencyEmit`) as a depth-general composition of a membership-adjacency half and an ordering half, with the emitted bytes pinned to the committed golden the prover loads. The custom effect — a cell program whose domain constraints are proven in an external sub-proof — resolves to a descriptor carrying the recursive-proof-binding constraint kind (`DescriptorIR2.ProofBind`): the row's proof-commitment and program-key columns are published as public inputs (`EffectVmEmitRotationV3.customPiExposure`), and the per-turn fold ties them to the folded sub-proof leaf. That binding is a theorem on the same floor as every other effect — a

verifying aggregate forces the published commitment to be backed by a verifying sub-proof whose program key is uniquely determined, and a forged commitment with no backing sub-proof makes the aggregate unsatisfiable (`CustomBindingFromFold.custom_binding_from_fold`, `CustomBindingFromFold.custom_companion_grounded`).

There is no unproven path beneath this coverage. The hand-rolled STARK engine is deleted from the tree; the descriptor prover over the byte-pinned registry is the only production proving path, and a resolver miss is a typed refusal, never a route to a hand-authored circuit. A repository gate keeps the coverage claim from rotting: `circuit-prove/tests/law1_enforcement_gate.rs` scans every source file of the two circuit crates for constraint algebra in each of the three dialects it can take — symbolic builder calls, evaluation closures, and constraint-expression data — against a frozen per-file baseline. A new file containing constraint algebra fails the build; a listed file that grows fails the build; shrinking is always allowed. The baseline's remaining entries are classified rather than amnestied: interpreters that evaluate Lean-emitted constraints, a proved-faithful encoding lowering, drift-detector twins the emitted paths check against at build time, and the user-facing predicate grammar. A hand-authored constraint therefore cannot re-enter a deployed circuit without turning the build red.

9 The firmament: one capability across a distance

The realization of Section 7 is a userspace service today, but its capability model does not stop at the process boundary. The **firmament** is the substrate that holds applications, gives them a deterministic ground to run on, and grants them capabilities under a single abstraction whose two backings — a local microkernel object and a distributed cell — are the **same** capability seen at two points on a distance parameter. This section is that unification and the strong properties it earns at the near end.

9.1 One capability, a distance parameter

A firmament capability is the handle of Section 3 — a `(target, rights)` pair an application holds, invokes, attenuates, or delegates — and its target is one of three backings. A **local** target is a microkernel kernel object (a capability slot, an endpoint, a frame): the invocation is a syscall, revocation is synchronous, and the rights are kernel rights. A **distributed** target is a cell on a (possibly remote) federation: the invocation is a turn through the executor, revocation is the group-key epoch lift of Section 7, and the rights are the grant attenuation of Section 3. A **surface** target is a cell rendered as a window (Section 10); a window **is** a capability over a cell, so it resolves by the same executor path as a distributed target.

The application does not see which backing it holds. It holds a capability and invokes it; the firmament routes the invocation. The unification is not a slogan but a theorem: the attenuation decision is the same `granted ⊆ held` gate regardless of target (`Firmament.attenuate_decision_backing_agnostic`), a widening grant is rejected at **every** backing, the surface (window-share) case included (`Firmament.no_amplification`, `Firmament.no_amplification_surface`), and a surface is provably another point on the same distance axis as a distributed cell, resolving through the same gate and the same verb (`Firmament.surface_is_another_point_on_n`, `Firmament.surface_gate_eq_distributed`). The verbs an application uses — invoke, attenuate, delegate — do not depend on the distance

(`Firmament.verbs_independent_of_n`); only the **bounds** on the operations do. The Lean development is the data-refinement obligation for a runnable bridge crate (`sel4/dregg-firmament/`) that path-depends on the genuine cell and turn semantics, so its local mint and its distributed delegate gate on the same attenuation predicate the kernel proves, with both polarities witnessed.

9.2 The single-machine principle: what collapses at n = 1

The honest bounds on a distributed capability are **distance** bounds, parametrized by the number of machines `n` the target is spread across. The firmament is the place where `n = 1` — everything is on one machine — so those bounds collapse to strong local properties rather than weakening into a degraded special case:

operation	`n > 1` **(distributed)**	`n = 1` **(firmament)**
revocation	eventual — the epoch lift must propagate	immediate — synchronous, dead when the call returns
checkpoint	a consistent cut across machines, possibly stale	a consistent checkpoint of one domain's state, atomic
commit	quorum / finality latency	synchronous — one durable write
agreement	FLP-bounded, needs a round	trivial — one machine is its own quorum

Table 6: The single-machine collapse. The same capability model; the distance parameter pinned to its minimum turns the distributed bounds into the strong local ones.

The collapse is mechanized: the distributed bounds equal the strong-local bounds at `n = 1` and relax above it (`Firmament.distributed_collapses_at_one`, `Firmament.bounds_relax_above_one`), with the protocol-level precedent for revocation specifically (`Distributed.Revocation.single_machine_collapse`). This is the single-machine principle made architectural: `n = 1` is not a crippled subset but the **collapsed limit** of one model. A local deployment is a first-class deployment, and the moment one of its programs invokes a capability whose target lives on another machine, the same handle routes over the wire and the program flows into distributed operation with no seam — the same verbs, the same receipts and proofs, only the bounds relaxing as `n` rises. The same binary that runs at `n = 1` scales out without a rewrite, because there was only ever one model.

9.3 The deterministic ground

The firmament's second job is to make the applications it holds **deterministic and reproducible** — pure, with no hidden nondeterminism and no path to assert a transition the kernel did not produce. The enforcement is structural rather than a matter of application good behaviour. An application holds **only** the capabilities the firmament minted for it, so ambient authority is absent: it cannot reach a clock, a random source, another application's memory, a device, or the network except through a capability handed to it, and a value like time or randomness it does need is supplied as an **input to its turn**, recorded in the receipt, so it enters the replayable transcript rather than as an ambient draw. Every action is a turn through the verified executor; re-running the recorded turns from a checkpoint reproduces the exact state, and that determinism is checked, not trusted, because the executor is the verified entry of Section 7.

Checkpoint and restore are this determinism made portable. A snapshot is a checkpoint plus an overlay — a full state at a height cut, plus every cell post-state committed after it — and the recovery equation `recover = checkpoint ⊕ overlay = replay` is a theorem for **any** cut, so where one checkpoints is free (`Distributed.CrashRecovery.recover_eq_replay`). The integrity tooth is a claimed root carried with the snapshot and bound to the chain's finalized commitment: the shipper never ships a snapshot whose reconstructed root differs from its recorded root, the joiner recomputes the root from `checkpoint ⊕ overlay` and refuses a mismatch, and a verified restore checks the claimed root against a root it already trusts — so a server cannot ship a self-consistent snapshot of a **different** ledger. This is orthogonal to the lifecycle **seal** of Section 7: seal pauses an application in place (reversibly, by `unseal`); a snapshot ships its state with an unforgeable self-attestation. A transparent move of a running application is seal, ship, restore-verified, unseal.

9.4 Status

The section's claims divide into theorems and running artifacts. The theorems are the capability unification, the `n = 1` collapse, and the recovery equation, each cited above to its Lean name. Three artifacts run.

The capability bridge runs against the real executor. The bridge crate (`sel4/dregg-firmament/`) routes local and distributed invocations through the attenuation gate the Lean development proves, and its tests exercise both backings, including boot, isolation, and surface migration.

The snapshot model and its root tooth are live in the durable store (`persist/src/snapshot.rs`). The shipper records the reconstructed root with the snapshot; the joiner recomputes the root from `checkpoint ⊕ overlay` and refuses a mismatch.

The executor protection domain runs. The verified entry the node calls — `dregg_exec_full_forest_auth`, the admission gate over `execFullForestG` — commits a turn inside a seL4 protection domain booted under `qemu-system-aarch64` and emits the accepted receipt over serial. The boot evidence and the build pipeline that produced it are in-tree (`sel4/dregg-pd/executor-rootserver/`, `sel4/dregg-pd/executor-pd/`): the proof closure recompiled to ELF under `leanc`, the Lean runtime rebuilt from the toolchain's sources with its event-loop dependency excised, and GMP cross-built for the target. The same executor is a Microkit protection domain in the multi-PD assembly (`sel4/dregg.system`); its capability partition grants the turn-input region read-only and the commit-output region read-write, and no device capability. An ingress domain accepts a signed turn over TCP, verifies the Ed25519 signature at the edge, and stages only an accepted turn into the executor's input region (`sel4/dregg-pd/net-client/`).

The device edges that remain, surveyed at the current tree: the durable-store domain holds its seat in the assembly but not yet a block-device capability, so durable commit without leaving the assembly — the store backend over a raw block capability — is open; the display domain scans out as the sole holder of the display device capability, and the verified per-viewer projection as that domain's render path is open (Section 10); the verifier domain enforces the one-way executor-to-verifier edge through its capability partition, and on-device checking of the deployed descriptor proofs is open (Section 11). Section 11 takes up the protection-domain partition in full.

10 deos: the agentic desktop

deos is the firmament made visual. It is the userlayer a human or an agent touches — windows, the surfaces they render, the compositor that draws them — built so that every visual and interactive primitive reduces to a kernel theorem. It adds no new trust: a window is a capability, an interaction is a turn, and what a viewer may see or do is decided by the capabilities they hold. The mathematics is the firmament's existing mathematics — attenuation, the admission gate, the receipt chain, unfoolability — restated for pixels, affordances, and rehydration.

10.1 A window is a capability

A surface is a `(target, rights)` capability over a cell (Section 9). A window therefore confers no authority beyond its rights, and a view- or notify-only surface confers no Granovetter edge at all (`Deos.viewSurface_confers_no_edge`, `Deos.notifySurface_confers_no_edge`): showing someone a window does not hand them the power to act through it. Narrowing a surface to fewer rights cannot amplify (`Deos.surface_attenuate_no_amplify`, an instance of the Section 3 attenuation law). This separates deos from a conventional desktop at the foundation: the right to see is distinct from the right to act, and both are capabilities.

The interaction model is declarative and server-rendered in the web's style, with the ambient-authority soup removed. A cell publishes named, typed **affordances** — effect templates — and an interaction is a verified turn: the "button" is a capability-gated effect, the rendered "fragment" is the attested post-state surface, and **who may press it** is decided by held capabilities rather than a session cookie. An agent fires only the affordances its capabilities authorize — the same `required ⊆ held` gate (`Deos.fire_authorized_iff`) — and the post-state surface binds the attested root of the turn it produced (`Deos.firedSurface_binds_attested_root`). Progressive enhancement becomes **progressive attenuation**: projecting a surface for a viewer is monotone in their authority (`Deos.projectFor_monotone`), so two agents see exactly the affordances their respective capabilities allow over one surface.

10.2 Per-viewer non-interference

Because a surface is projected per viewer, deos can state the property a cross-domain compositor usually only **trusts** its rendering process to provide: non-interference. A low-authority viewer's render is a function of the low-authorized state **alone** — changing a cell that viewer cannot see leaves their view bit-identical (`Deos.noninterference`, `Deos.hidden_change_invisible`), a hidden cell is structurally absent from the projection rather than merely undrawn (`Deos.hiddenCell_absent`), two viewers diverge by exactly their authority (`Deos.divergence`), and vision is monotone in capability (`Deos.vision_monotone`). What a viewer sees is determined by exactly the fragment inside their capability — the information-flow sibling of rehydration confinement below.

The compositing underneath is an algebra with the guarantees a windowing system's correctness rests on, here proved rather than assumed. Damage is exact — a present dirties exactly its declared regions and nothing outside them (`Deos.present_damage_exact`); paint is order-free on a well-formed scene, so z-order does not affect the pixels (`Deos.paint_order_independent`); and editing one window cannot perturb another's pixels, the compositional dual of non-interference (`Deos.render_frame_property`). Re-rendering is functorial over the per-viewer projection: re-

rendering after a state update equals updating the rendered surface, the central web-framework guarantee (`Deos.rerender_square`).

10.3 The rehydratable surface

The primitive this architecture makes possible is the **rehydratable surface**. A deos snapshot — a "screenshot" — is a frame of the certified compositor over the witness graph, and what it actually embeds is a **sturdy reference behind a membrane**: a persistable, attenuable capability that, when the image is opened, re-attaches a live — or faithfully replayable — interactive surface. It is not a faithfulness proof retrofitted onto a dead pixel grid, which would ask a viewer to instantiate arbitrary author state. The fidelity is upstream and structural: the frame came out of a compositor whose render is itself a verified projection over the witness graph, so it can only draw what the graph authorizes. The snapshot rehydrates because it was never a dead artifact — it is a paused camera on a witnessed scene.

The membrane makes rehydration **relational**. Two agents opening the same snapshot do not reconstruct identical surfaces; each renegotiates, across the membrane, the slice their capabilities authorize. Reshares compose attenuation across hops — re-sharing $A \to B \to C$ confers on C no more than B held, for chains of any length (`Deos.reshare_chain_attenuates`, `Deos.reshareN_attenuates`) — and a widening reshare is darkened, not granted (`Deos.reshare_refuses_amplification`). Sharing a surface stops being "I leaked my session" and becomes "I extended a revocable, attenuated, per-viewer right to re-view." The snapshot is a lossless per-viewer handle, not a lossy thumbnail: it re-expands faithfully and under attenuation (`Deos.snapshot_roundtrip`, `Deos.snapshot_roundtrip_attenuated`).

10.4 The liveness type is the confined fragment

The cost of "live **or** replayed" cannot be negotiated away — if the original contexts are gone, replay fidelity is bounded by how deterministically the witness graph captured the scene's nondeterminism — so the membrane **types** every reacquisition with one of three values: $\{\text{Live}, \text{ReplayedDeterministic}, \text{ReconstructedApproximate}\}$. The type is not a label of good intent; it is a **proven confinement readout**, computed from what the witness graph actually attested. The central theorem is that *ReplayedDeterministic* is **exactly** the confined fragment: for a non-live context, it classifies as ReplayedDeterministic if and only if every interaction it made was a witnessed, attested turn (`Deos.replayedDeterministic_iff_confined`). A context whose nondeterminism all flowed through attested turns can be replayed deterministically by construction (`Deos.replayedDeterministic_replays`, riding the receipt-chain tamper-evidence of Section 4); a context that reached for nondeterminism **outside** the membrane — an unwitnessed clock, an ambient random draw, an un-attested external call — is intrinsically *ReconstructedApproximate*, because the very thing that made it nondeterministic was never captured. The classifier partitions non-live contexts with no third bucket (`Deos.reconstructedApproximate_iff_unconfined`), and a structurally invalid attestation — one lacking quorum — cannot launder a context into the replayable class (`Deos.invalidAttestation_not_replayed`). So the liveness type is a readout of confinement, not a claim of honesty: the system cannot misreport which kind of true an opened image hands a viewer, because the classification is derived from the attested record rather than asserted over it.

10.5 What runs

The Lean development is kernel-clean throughout: each of these is an existing kernel proof — attenuation, the admission gate, the receipt chain, projection — restated for surfaces. The single named hypothesis is the receipt-digest injectivity the replay payoff carries (the same floor carrier of Section 17), stated as a hypothesis, never an axiom and never an admitted goal. The Rust realization of the rehydration and affordance stack ships in `starbridge-web-surface`.

On the microkernel, the display edge boots. `dregg-graphical.system` is a seL4 Microkit image whose compositor-fb protection domain is the sole holder of the display device capability — the `fw_cfg` MMIO region — and of a 2 MiB DMA framebuffer. The domain configures the `ramfb` scanout over `fw_cfg`, draws into the framebuffer, and the scanned-out frame matches the written bytes pixel for pixel (`sel4/build/dregg-graphical.img`; `docs/desktop-os-research/GRAPHICAL-SEL4-BOOT.md`). The one-device-cap-per-domain discipline of Section 11 therefore reaches glass. What remains is the render path above it: the pixels that domain currently draws are a static splash, and the open piece is running the verified per-viewer projection as that domain's frame source, with the compositing theorems gating what reaches the framebuffer.

11 dregg on seL4: capabilities all the way down

The firmament's strong properties (Section 9) want a substrate that shares its thesis: capability security by construction, enforced by the machine. seL4 is that substrate. A dregg image on seL4 stacks two capability graphs — the seL4 graph isolates the protection domains, and the dregg graph mediates the cells inside them — so the deployment is capabilities all the way down. The stack is not a design sketch. The verified executor of Section 7 — the compiled Lean kernel the node calls — runs a real committed turn inside an seL4 protection domain, and a six-domain node assembly builds with that executor in its seat.

11.1 Two capability graphs, one thesis

The deployment is a small assembly of protection domains under the seL4 monitor (`sel4/dregg.system`). Each domain holds only the capabilities its role requires. The **executor** domain embeds the verified entry and holds exactly two memory capabilities: read on the turn-input region and read–write on the commit region. It holds no device capability. The **verifier** domain is pure compute over bytes; it holds no prover authority, no storage capability, and no network capability, and its only connection to the executor is a one-way notification channel. "A verifier never calls back into a prover" is therefore a property of the capability graph, not of code review. The **persist** domain is the designated seat of the storage-device capability and the only domain besides the executor that maps the commit region. The **net** domain is the sole holder of the network-interface capability and touches nothing else. The **ingress** domain runs the TCP/IP stack, checks a turn's signature at the edge, and is the sole writer of the executor's input region, so a badly signed turn never reaches the executor. An **application** domain (the directory cell below) holds only what its factory minted. Ambient authority is structurally absent, enforced by the memory-management unit and the capability-derivation tree rather than by convention.

The stacking is mechanized, with a stated scope. The Lean development transcribes the pure fragment of seL4′s own abstract specification — each definition headed by its l4v source

file and line range, pinned to l4v commit `e2f32e54`, with `derive_cap` pure-ised by threading the `ensure_no_children` verdict as a boolean — and proves that the transcribed derivation never amplifies conferred authority (`Firmament.SeL4Abstract.seL4_derive_cap_non_amplifying`). The one opaque branch, architecture-specific capabilities, is a named hypothesis; it is what an l4v arch-derivation proof discharges per architecture. The bridge theorem (`Firmament.SeL4Abstract.dregg_executor_cap_authority_grounded_in_seL4`) then derives dregg's own capability non-amplification — the cap-authority leg of the running-entry guarantee (Section 17) — from the seL4 lemma rather than re-proving it on the dregg side. The derivation runs under a named bridge assumption (`Firmament.SeL4Abstract.SeL4DeriveNonAmpBridge`) whose embedding-faithfulness obligations are carried as hypotheses, not proved; an l4v-to-dregg refinement would discharge them. The authority relabelling between the two vocabularies is total and injective on the eight seL4 authority constructors dregg uses (`Firmament.SeL4Abstract.alpha_total_iff_used`); the remaining four project into other parts of the model (memory access into the memory-checking model, revocation into the registry, arch authority above dregg's level). So one leg is grounded, by a named reduction; this is not a refinement of dregg against l4v. Within that scope the two graphs are one discipline at two scales: the same `granted ⊆ held` law that governs a delegated cell capability governs a derived seL4 capability, which is the firmament's distance-parameter claim (Section 9) seen from the substrate.

11.2 What boots

The executor domain is the central boot. Getting the compiled Lean kernel onto the microkernel required rebuilding its runtime, and the port is itself a checkable artifact (`sel4/dregg-pd/executor-pd/WALL.md`). The verified closure rooted at the FFI entry is recompiled to ELF, excluding one metaprogramming leaf that exports no runtime function. The Lean runtime is rebuilt from the toolchain's own sources for a hosted musl target, with its event-loop library replaced by a stub — the pure entry takes bytes and returns bytes, so the turn reaches no socket, file, or timer. GMP is cross-built; a small shim maps the runtime's allocator onto musl `malloc`; the link closes with zero undefined symbols. On that runtime, inside a protection domain booted under QEMU, `dregg_exec_full_forest_auth` — the exported form of `FullForestAuth.execFullForestG` with admission — runs a committed transfer turn: the nonce advances, a 30-unit transfer moves conservatively across two cells, and a nullifier and a commitment register, with the same accepted receipt the host build produces (`sel4/dregg-pd/executor-rootserver/`).

The domain's cryptographic portals are real, not stubbed (`sel4/dregg-pd/executor-pd/crypto-floor/`). Poseidon2 over BabyBear matches the circuit's two-to-one hash; BLAKE3, the Poseidon2-derived nullifier, and the keyed MAC match the transcript primitives; strict ed25519 verification is the executor's own authorization check; the Ristretto255 value commitment is byte-identical to the cell's; the note box's authenticated encryption opens genuine ciphertexts and rejects tampered ones. The one portal that does not compute is proof verification, and it fails closed (an unverifiable proof rejects); the next subsection states why.

The partition above assembles into one bootable image with the verified executor in its seat (`sel4/build/dregg-real.report.txt` lists the executor's thread among the six domains). Fitting the roughly 280 MiB executor image under the Microkit tool takes a one-function patch mapping image segments with 2 MiB pages. In the recorded assembly boot the executor commits its turn and signals the persist and verifier domains, the persist seat observes the commit signal, the

network driver probes a real virtio-net device and brings the link up, and the directory cell enforces its membership list, rejects a stale compare-and-swap, and mints only the object shape its factory slot allows. A separate two-domain image welds the executor to a display domain: the receipt the executor writes drives a repaint in the viewer over a cross-domain notification, so a verified commit in one protection domain changes pixels owned by another (Section 10). The smaller bring-up rungs also stand: a minimal banner domain, a structural verifier domain exercising the bundle-in, verdict-out contract, and the banner domain retargeted to and booting on a second instruction-set architecture (riscv64).

11.3 Proof checking on the device

The verifier domain today boots, holds nothing beyond its notification channel, and acknowledges the one-way executor-to-verifier edge (`sel4/dregg-pd/verifier-stark/src/main.rs`). An earlier bring-up carried a hand-authored STARK engine into this domain byte-for-byte and proved, verified, and tamper-rejected a small arithmetization on the device. That engine was deleted repo-wide when the Lean-emitted descriptor prover became the single production path (Section 4), and the vendored on-device copy went with it. On-device proof **checking** is therefore not currently demonstrated. The production verifier — `verify_vm_descriptor2` over the emitted descriptors (`circuit/src/descriptor_ir2.rs`) — is a hosted workspace crate on the Plonky3 stack, and it has not been carried into the `no_std` domain. Until it is, the executor's proof-verification portal rejects rather than passes, and proof checking for the seL4 deployment happens off the device, by the same light client any other deployment answers to.

11.4 What remains

Recomputed at the current tree, the remaining work is four named items. First, on-device proof checking under the production prover: carry the descriptor verifier into the verifier domain. The deleted demo shows the port shape — the same carry that moved the hand engine — but the descriptor verifier's dependency set is larger. Second, the persist seat is a stub: it maps the commit region read-only and observes commit signals, and the storage-device capability with a durable store behind it has not landed. Third, an externally delivered turn end to end on the device: the assembly wires TCP ingress, signature check, staging, and the executor signal, but the recorded boots exercise the executor's compiled-in turn rather than one arriving over the wire. Fourth, the bridge assumption's embedding obligations remain hypotheses. Every domain named above boots; the four items name exactly the distance between that and the full deployment.

12 pg-dregg: verified durable state

The receipt chain of Section 4 is the truth and the database is a cache. pg-dregg makes that relationship a concrete deployment: dregg as a PostgreSQL extension in which **reads are ordinary SQL and writes are verified turns**. State exists in the database only as the post-image of a turn the verified semantics accepted. An application queries that state freely with `SELECT`; every mutation passes through the verifier. It is the embeddable realization — the verified executor of Section 7 reachable from inside a database engine — and it carries the kernel's guarantees into a place applications already live.

12.1 The spine invariant

The discipline is one sentence: reads are free SQL, and state mutates only through verified turns. A state row exists solely as the post-image of a verified turn, never by a bare SQL `INSERT`, `UPDATE`, or `DELETE`. Ordinary `SELECT` reads the materialized state with no proof obligation, because reading cannot violate a guarantee. Writes are the only thing the verifier need gate, and they are gated at the one door the schema exposes: the commit-log table, whose `BEFORE INSERT` trigger refuses a batch the chain discipline rejects. The same capability token of Section 3 authorizes both sides. A row-security policy that admits a read and the credential a write must carry evaluate the same admission decision, so the database's access control and the kernel's authority are not two systems kept in sync but one decision evaluated in two places.

12.2 The tiers

pg-dregg realizes the spine invariant in layers, each usable on its own and each adding one capability of the verified substrate to the database.

tier	**what it adds**
authorization	dregg capabilities as row-security policies: a policy admits a row only when a configured-issuer credential authorizes the action, verified offline against the issuer public key — the Section 3 admission decision, with no circuit and no network. The credential core is the same authorization library the node runs, anchored by a Lean–Rust differential
mirror	the node ships each verified turn's rows into read-only tables (turns, cells, capabilities, memory); applications query verified state as plain SQL joins, and the sole writer is the verified commit path, so the spine invariant holds
verified store	the commit-log table is the only door to state, and its before-insert trigger re-runs the chain check — turn N's post-root is turn $N\{+\}1$'s pre-root, ordinals dense — so a reordered, gapped, or substituted batch is refused by the database engine itself; a proof gate over the same store marks whole windows of turns proof-attested
embeddable executor	a database function runs a turn in-backend through the verified entry, writing the receipt and the post-state in one transaction — the database **is** the kernel, with turn and application data sharing one atomic commit

Table 7: The pg-dregg tiers. Each is a point at which the verified substrate enters the database; the lower tiers are circuit-free and offline.

The lower tiers need no prover: the authorization tier is a credential check compiled to a row-security predicate, and the mirror tier turns the database into a query surface over verified state with the node as sole writer.

12.3 The write path

The write spine has two halves. A database user calls a submit function whose insert is itself row-security-gated; the call records a pending intent in a queue, and postgres never executes

anything at that point. A drainer then processes each intent through four gates in order. It rechecks the acting agent's capability, consulting revocation, so a token revoked after enqueue is refused. It produces the post-image through the executor seam. It admits the produced batch onto the durable head by the chain check, and a batch that does not chain never moves the head. It materializes the post-image rows and resolves the queue row — executed with the receipt hash, or refused with the reason — in one logical commit, so the submitter learns the outcome by reading its own row.

12.4 The proof gate

The verified store's per-row chain check is structural: it proves the stored sequence is consistent, not that each turn executed correctly. The proof gate adds the second half. A recursive whole-chain proof (Section 4) over a span of finalized turns is presented as serialized bytes, and a set-returning function verifies it and emits one row per covered ordinal, each tagged proof-attested. A consumer joins those rows against the turns table to distinguish the proof-attested prefix from the merely chain-consistent one.

The serialized transport carries the verify-sufficient subset of the fold: the root proof, the chain-binding proof, and the public scalars. Prover-only chaining data never crosses the SQL boundary. In the proof-linked build the gate runs three checks: the verification-key pin against a caller-held anchor, the carried publics against the binding proof, and the root batch verification. The verifier it links is the recursion verifier of the descriptor stack — the same check the light client makes — built without the executor or the Lean runtime. Both the admit and the refuse directions are exercised by a test that folds a real turn chain, verifies it through the SQL-side transport, and then confirms a relabeled public and a foreign verification key are refused. The default, circuit-free build decodes the transport and attests nothing: a proof gate that cannot verify reports a window as unattested, never as attested.

The producer that folds newly finalized turns into attestation rows is in place as machinery around a seam: window discipline, watermark resumption, and a check that binds the fold's claimed coverage to the exact window it was handed are tested against a deterministic stand-in, and the real circuit fold plugs in where the circuit is linked. Two edges remain named. The live node-side wiring of that real fold is one. The transport's state anchors are the other: it carries the single-element head root rather than the full eight-element anchor, and a genuine wide anchor is refused rather than mis-attested until the transport is widened.

12.5 The embeddable executor

The deepest tier embeds the verified executor itself. A database function takes a turn, runs it in-backend through the compiled Lean entry of Section 7, and writes the receipt and the post-state in the same database transaction as the application's own writes. The runtime it embeds is the single-threaded, fork-safe configuration of the Lean runtime: a private allocator that never overrides the backend's own, lazy task management that runs spawned tasks inline, and an initialization that omits the event-loop thread. Initialization happens lazily on the first produced turn, in the worker the engine forked, never in the postmaster; a failed initialization refuses the turn rather than committing unverified state. The executor tier is linked only when enabled, so the default build never sees it.

This tier runs, and the extension's own test suite exercises it inside a live backend. One test drives the producer directly: the runtime initializes in-backend, `execFullForestG` executes, and a conserving transfer commits with post-balances read back from the executor's decoded state. A second test drives the full SQL path: a submitted turn is drained through the four gates, executed by the embedded verified executor, and lands as a committed turn row, with the test first asserting the runtime is live so that a drained turn is one the executor decided rather than a stand-in fold. When the Lean executor is linked it wins producer selection; a pure-Rust executor kept at documented parity with the Lean specification is available as a lighter producer, and the default build retains a deterministic stand-in so the gate plumbing is testable with no executor in the build.

This is where the kernel's theorems land directly in the data path. Conservation, non-amplification of delegated capabilities (`Spec.gen_conferral_is_attenuation`), nullifier uniqueness, and authenticated state-root evolution hold of the rows because the function that wrote them is the verified executor; the post-state is verified by construction rather than re-checked. One seam remains and is named: the extension does not link the turn and cell marshalling crates, so the in-backend producer cannot decode the submitter's signed envelope into the executor's wire turn. It synthesizes a conserving transfer instead and takes the executor's verdict and post-state as authoritative. Full in-backend decoding of an arbitrary submitted turn is the open edge (Section 19).

The runtime boundary this tier crosses is the one the seL4 port also crossed (Section 11). A postgres backend is a forked, single-threaded process under the engine's error handling; a protection domain offers no threads and no ambient services at all. Both hosts require the Lean runtime to run with no worker thread and no event loop, and the same excision — omit the event-loop initialization, keep the allocator private — serves both. One executor, two hosts, one boundary.

12.6 Scope

pg-dregg is a mirror, a verified-write gate, and an embedded light-client backend for the database, not a full node. The authorization and mirror tiers are circuit-free and live. The verified-store chain check is live, and the proof gate verifies real recursive folds in the proof-linked build. The embeddable executor runs the verified step in-backend under the extension's tests. The named edges are full in-backend decoding of an arbitrary submitted turn, the live wiring of the node-side proof producer, and the wide-anchor transport. Federation across databases is structural: replication is single-writer fan-out, and a subscriber re-runs the chain check on the replicated rows, refusing a tampered or reordered stream and confirming it holds the whole published prefix rather than a truncation. Section 19 states the in-progress edges as checkable facts.

13 The game portfolio

The factory userspace of Section 7 has a first product family: games. A game turn is the model's opening sentence with nothing adapted — the exercise of an attenuable, proof-carrying token over owned state, leaving a verifiable receipt. The receipt is the run. The same primitive serves the player, who gets a fair game no operator can falsify, and the author, who gets an engine whose contract is a kernel theorem rather than server code; the flagship game is the first client of

the platform schema it motivates. This section states what each maturity level in the portfolio is: what is proved in Lean, what is exercised by tests, what is deployed, and what is coded behind an operator switch that has not been flipped.

13.1 Three games, one shape

The Descent (`dreggnet-offerings/src/daily_descent.rs`, played through `discord-bot/src/commands/descent.rs`) is a daily dungeon crawl. Each day one world is generated from the drand `quicknet` randomness beacon, so every player plays the same seed and no one — operator included — can grind a favourable dungeon; the beacon check is a real BLS pairing verification against the pinned group key, and fetching the day's round from a drand node is the named client seam. The stakes are enforced by the executor, not the narrator. A lethal position routes to a committed defeat passage: the hit-point floor refuses every other move, a write-once `downed` flag ends the run, and a lost run re-verifies by replay exactly as a won one does (`dungeon-on-dregg/src/bloodgate.rs`). A d20-versus-difficulty gamble gates a shortcut, with the roll bound into the receipt so a forged pass is caught on re-verification, and a shared write-once slot prices an either/or choice. Opt-in hardcore permadeath is write-once-final over a persistent character (`dreggnet-offerings/src/character.rs`, durable in the bot's sqlite store), and a weekly bracket advances only on a verified win (`dreggnet-offerings/src/descent_tournament.rs`).

The narrator is a language model, and the division of authority is strict: the model proposes, the verified rules dispose. Hit points, inventory, and dice outcomes are cell fields governed by the cell's program on the executor path, and the narrator holds no capability to write them outside an admitted turn. A narration that contradicts the committed state changes nothing, so the game master cannot misreport the world it narrates. Attesting the narrator itself — that a disclosed model produced the prose — is designed against the attestation carrier and has a named operational remainder, a live pinned-notary session (`docs/GAME-STRATEGY.md`).

The other two games are board games whose rulebooks are Lean definitions. **automatafl** is a board game of indirect control: players place attractors and repulsors, moves are resolved for conflicts, and a neutral automaton takes one rule-determined step. **Multiway tug** is a hidden-hand card game whose plays open against a committed hand. For both, the deployed circuit's admission relation is proved equal to the rulebook's step function, stated over the staged form the circuit actually computes rather than over a restatement of the rules.

13.2 The rulebook theorems

For automatafl, the staged circuit admits a triple exactly when the successor board is the rulebook's turn function applied to the old board and moves (`Games.Automatafl.airAutomatafl_iff_applyTurn`); a successor that relocates a piece anywhere the rules do not has no satisfying witness (`Games.Automatafl.airAutomatafl_forged_refused`); the circuit admits at most one successor per position (`Games.Automatafl.airAutomatafl_functional`); and the in-circuit conflict-selection table is proved to match the reference resolution (`Games.Automatafl.conflictResolve_pair`). The packaged circuit discharges the abstract refinement obligation rather than leaving it a bare hypothesis (`Games.Automatafl.concreteAutomataflAIR_refines`). For the tug game, a play is admitted exactly when it is legal, its cards open by membership proof against the committed hand root, and the successor is the rulebook's (`Games.MultiwayTug.airPlay_iff_applyAction`);

after a play, the committed root of the remaining hand is the commitment of the hand minus the played cards, so replaying a spent card fails membership under the new root (`Games.MultiwayTug.remaining_root_updates`); consecutive leaves compose as consecutive rule steps (`Games.MultiwayTug.airPlay_chain_are_applySteps`).

The statements carry their remainder explicitly. The arithmetization soundness of the low-level gadgets — that a satisfying witness forces the move-resolve and automaton gadgets to compute their reference stages, and that Merkle membership implies hand membership — is carried as named hypotheses (`Games.Automatafl.MoveSound`, `Games.Automatafl.StepSound`, `Games.MultiwayTug.MerkleSound`), in the same discipline as the engine carrier of Section 4: hypotheses, never axioms, so the axiom audit stays on the kernel triple while the obligation remains visible in the statement. Discharging them is the deployed circuit's job, not the model's.

13.3 The tests against the statements

The Rust game crates are tested against these theorems rather than beside them. A fast refinement battery (`dregg-automatafl/tests/refinement.rs`) drives the built circuit against a reference oracle that mirrors the Lean rules and their `#guard` cases: the circuit accepts the honest successor and rejects a wrong successor, an invalid move, and a forged conflict resolution. The proving battery (`dregg-automatafl/tests/prove_fold.rs`, minutes-long and run on demand) takes each staged circuit — the automaton step, the single-move apply, the two-move resolution, and a sealed-move stage whose Poseidon2 commit-and-reveal is opened inside the circuit — through the deployed path of Section 4: the circuit proves as a recursion-foldable leaf, the in-circuit commitment byte-matches the host binding, the leaf binds into a `Custom`-effect turn, a turn chain folds through the deployed recursive prover, and the light client's `verify_history` accepts. The negative directions are driven at the same depth: a forged automaton step, a forged apply, a reveal that opens a move other than the committed one, and a spliced final root each fail to produce an accepting artifact.

The staging has a measured boundary (`dregg-automatafl/tests/size.rs`). The two-move and single-move stages run the automaton gadget a second time on the resolved board, and at board size five their trace widths (1178 and 1411 columns) exceed the prover's 1024-column ceiling; they fit and prove at board size three. The full 11-by-11 board and general N-move resolution are the named residuals, to be closed by a segmented board-read scan. Complex mechanics of this kind ride the custom-leaf path, not the standard per-effect descriptors.

13.4 Cheating, today and next

On the deployed surface, no-cheat is by replay: a submitted run carries its public inputs, the server re-executes it on the same executor path, and the leaderboard admits only runs that re-verify. Replay already suffices for the central claim — the operator cannot misreport a hit-point total, and a forged roll or resolution is caught — but the checker must re-run the game. The labeled upgrade is the portable STARK proof: a completed run checkable by a stranger from its receipts alone, with no re-execution and no trust in the serving host. The fold tests above exercise exactly that path end-to-end for the board-game leaves, so the upgrade is a change in what the submission carries, not new proving machinery.

13.5 What is deployed, and what awaits a flip

The public surface is the games web server behind a Tailscale funnel hostname (`deploy/games/RUNBOOK-FUNNEL.md`): five games and the replay leaderboard, served from one host as a user service unit that has survived a host reboot. Its receipt-anchoring leg is currently down — the devnet node the surface anchored runs to was hand-run with an ephemeral data directory and its ledger did not survive a host power-cycle — so a submitted run ranks in-process and anchoring fails closed until an operator brings up a durable node.

The payment loop is coded and tested but not live. The `dregg-pay` crate implements a custodial backend: per-user deposit addresses by hardened SLIP-0010 derivation from one seed, a payment watcher, a per-user run-credit ledger idempotent per payment reference, and a treasury sweeper. Its economics are dual-asset: USDC is the fuel that real-AI runs draw down, $DREGG accumulates in an illiquid treasury pile, a run costs $0.10 by default, paying in $DREGG earns a 20% discount through a price oracle, and an over-the-counter leg sells pile $DREGG for USDC at a 10% discount. The end-to-end suites pass on mock chains, including the liquidity-governance vote that authorizes a signed pile-to-fuel swap. The Discord bot consumes the backend: a buy-credits command issues the caller's deterministic deposit address, polling credits payments idempotently into a sqlite-persisted ledger, and a paid run debits one credit only after a successful narration under a per-run dollar budget, falling back to the free tier on an empty balance (`discord-bot/src/pay.rs`). Nothing mainnet is hardcoded; the bot defaults to a mock watcher on devnet parameters, and the deployed surface's environment sets no payment variables at all. Going live is an operator configuration change — mint, treasury, seed, network — plus a custody watcher/sweeper service that does not yet run, and no $DREGG has been accepted for a service on the deployed surface. The design endgame is protocol-native settlement, a run budget as a conserved transfer balance with no operator custody; the custodial backend names itself the bridge to that.

The token's role here is fixed by a locked policy: $DREGG buys services — narration credits, hosting, cosmetics, entry — and never buys power, features, or yield; the leaderboard confers standing, not income. The asset itself, its supply discipline, and the fee mechanics are the subject of the paper's treatment of the system's own economics.

13.6 From one game to an engine

The platform half of the portfolio is a schema between authors and cell programs (`dregg-schema`). An author declares typed component archetypes — stats as bounded fields, resources as monotone counters, identity as write-once, collections as heap fields — and an allocator lowers them to a slot and heap layout by translation validation: the search is untrusted, the output is checked against a legality discipline under which an ill-aligned layout cannot be constructed, and the result is a generated cell program and a typed API. The Lean-proved legality obligation over this allocator, and the leaf refinement from schema output to the fold, are its named next resolution. Play UX rides the capability model directly: a session key is a caveat-bounded delegation of the player's play capability — attenuation, no new trust model — whose paymaster binds to the same run-credit ledger, so an out-of-credit move commits nothing (`dreggnet-offerings/src/session.rs`); a session resumes by replay of its reproducible public input, never from a trusted blob.

The portfolio's maturity gradient is uniform and deliberate. The rulebooks and their circuit refinements are proved; the proving path is exercised by tests at deployed depth; the play surface is deployed with replay as its verifier; the payment loop is code behind an unflipped switch. What

a player exercises today is already the object the kernel theorems govern. The labeled upgrades change how a run is checked and how it is paid for, not what a move is.

14 Interchain settlement and proof-of-holdings

dregg networks proofs, not tokens. A bridge answers "did this happen on that chain?" by trusting a validator set; dregg answers the same question with a proof the asking side checks itself. The exchange runs in two directions. Inbound, dregg acts as a light client of a foreign chain: a holder proves a balance there and receives governance weight here, and nothing moves. Outbound, a foreign chain acts as a light client of dregg: its on-chain verifier checks a proof of a dregg state transition and settles. The outbound direction is the light-client theorem exercised across a chain boundary (`RecursiveAggregation.light_client_verifies_whole_history`): the aggregate a stranger can check with one verification is the same object whether the stranger is a phone, an auditor, or a contract on another chain. The inbound direction is the same discipline pointed the other way — dregg admits a foreign fact only against a verified consensus proof, never against an attestation it cannot check.

14.1 Inbound: proof of holdings

A holder proves that their own token account on a foreign chain held a balance at a finalized point, and is granted vote weight equal to the proven balance. Custody never moves: no lock, no escrow, no wrapped token. On Solana the production verifier (`bridge/src/solana_holdings.rs`) checks, fail-closed: token-program ownership and mint binding of the holder's account; a stake table **derived** from proven bank-state accounts — stake, vote, and the stake-history sysvar with the warmup/cooldown effective-stake curve — admitted only against a governance-pinned weak-subjectivity anchor, never supplied by the caller; an authorized-voter-bound, stake-weighted $\geq \frac{2}{3}$ Ed25519 tally; recomputation of the bank hash; a 16-ary accounts-hash inclusion proof of the holder's account; and a bounded-anchor proof-of-history policy. The trust model is the standard light-client one: trustless over a pinned anchor. With no anchor configured the path refuses; there is no fallback to a caller-supplied table.

The holder's foreign key binds to a dregg voter identity by signature, not by transfer: strict Ed25519 over a domain-separated message for Solana wallets, EIP-191 `personal_sign` with secp256k1 recovery for EVM addresses, and pubkey-carrying secp256k1 with the address-hash equality as the load-bearing check for Cosmos addresses (`dregg-governance/src/holding_weight.rs`). Chain-shape dispatch prevents a binding from one family being replayed into another. Double counting is closed on two axes. Each poll pins one finalized snapshot height per chain, so moving tokens between accounts and re-proving fails against the pin. Each (poll, chain, holder, asset) tuple is a consume-once nullifier, so re-presenting the same holding is refused; a refusal never consumes the nullifier, and a weight too large for the ballot domain refuses the cast rather than saturating. The same holder on two chains is two nullifiers, and both count — the holdings are distinct facts.

The weight verdict itself is a Lean function. The Rust pipeline performs the pre-checks (consensus tier, owner binding, positive amount) and then calls the exported core `ProofOfHoldings.grantWeightCore` over FFI; a missing core is an error, never a Rust reimplementation. `ProofOfHoldings.grantWeightCore_eq_grantsWeight` proves the exported de-

cision realizes the specification, and `ProofOfHoldings.weight_backed_and_noncustodial` is the guarantee: a granted weight is backed by a consensus-proven holding of at least that weight at a finalized slot, and the grant leaves the chain state definitionally unchanged (`ProofOfHoldings.grant_preserves_custody`). The model generalizes past Solana: the chain-agnostic statement is `ProofOfHoldingsGeneric.weight_backed_and_noncustodial_generic`, and the weighted **decision** is fold-compatible — `HoldingWeightedTally.decision_backed` states that a passing outcome's cleared weight is exactly the sum of consensus-proven, non-custodial holdings, evaluable over independently verified ballot segments. The per-chain trust dials collapse onto one fail-closed ordinal whose verdict is likewise the proven object (`InterchainAdapterDecision.reachedConsensusCore_correct`): the lowest, uninitialized rung refuses — the polarity whose inversion drained the Nomad bridge.

The verifier has an adversarial history. Its predecessor accepted a caller-supplied stake table, which is a weight forgery: a one-key attacker table clears its own supermajority. An audit found this in 2026-07; the entry is now compiled only under test gates, and the production tests drive both polarities — a supermajority over the holder's account verifies, while a sub-supermajority tally, a wrong mint, an unauthorized voter, a tampered accounts hash, and an attacker-supplied one-key table are each refused (`bridge/tests/solana_holdings.rs`).

Three boundaries define the current resolution. A live-feed rung now ingests a real SPL holding, stake and vote accounts, and the StakeHistory sysvar over RPC from `solana-test-validator`, then proves the holding end to end through the same anchored verifier (`bridge/src/solana_feed.rs`, `bridge/tests/solana_local_e2e.rs`). This is live transport against a local validator, not mainnet provenance: the mainnet snapshot/geyser source and its operator-pinned anchor remain unbuilt. Weighted ballots still land in a host-side ballot box whose one-vote and quorum gates are a hash set and a comparison; the verified vote engine has no landed weighted-cast path, so the proved object is the weight verdict, not the box it enters. Finally, consensus verification is off-circuit — re-executing-verifier grade, not folded into an AIR — so a succinct attestation of the holding proof is future work, not a present claim.

14.2 Custody has three modes

The holdings path above is one of three custody modes, and the distinctions carry the security claims. **Native dregg assets** are self-custodied and trade freely under the kernel's conservation law. **Holdings for governance** are proved, not locked: a read-only weight that moves nothing. **Foreign assets for trading** are different: to make an off-chain asset spendable inside dregg, the holder locks it into a vault and dregg mints a 1:1 mirror; exit burns the mirror and releases the lock. The lock is what prevents a double-spend, and it is custody — surrendered to a program that releases only on evidence, never to a custodian or a validator set, but custody nonetheless. The trading mode is proof-gated custody, not non-custodial, and the governance mode is the participation front door; the vault exists for importing spendable value and posting bonds.

14.3 The vault

The Solana side is a native token program (`solana-lock/`): a lock transfers into a program-derived vault and writes a lock record whose identifier is the record's derived address. The dregg side mints against that lock through one committed gate, `TurnExecutor::bridge_mint_against_lock` (`turn/src/executor/bridge_ledger.rs`): evidence that is not consensus-verified is refused before

any state changes, a domain-separated lock nullifier is consumed against the committed nullifier set so two relayers cannot mint twice from one lock, and the mint draws against an independently recorded escrow leg under the invariant that live mirror supply never exceeds the currently locked amount. Given truthful lock evidence, an unbacked mint is structurally impossible; the two-relayer race is refused in a committed-state test (`bridge/tests/committed_double_mint.rs`).

Inbound lock evidence arrives at one of two labeled trust levels. The trusted-oracle level — a threshold attestation by a configured signer set — is the production slice. The trustless level (`bridge/src/solana_trustless.rs`) verifies the lock with the same consensus-anchored machinery as the holdings path: a derived stake table, an authorized-voter-bound tally, bank-hash recomputation, and inclusion of the vault account, with acceptance and five refusal polarities under test (`bridge/tests/solana_lock_trustless.rs`). That leg has verified only fixture-built clusters; it has never verified real mainnet consensus, and the succinct in-circuit form of the statement is named, not built. Release is narrower than lock verification on every path: unlocking on Solana always requires an M-of-N Ed25519 oracle attestation, so the outbound leg of the vault is oracle-custodial even where the inbound leg is trustless.

The three value-path defects found in review are closed in the verifier. Value release now additionally requires a stake-weighted, authorized-voter-bound rooted attestation; an exact-slot vote without a tower root yields `SlotNotRooted`. Stake derivation compares supplied effective stake with the cluster floor carried by the proven StakeHistory sysvar, so omitting stake accounts cannot shrink the denominator below that floor. Epoch rotation uses the same `tally_authorized` binding as ordinary votes. The acceptance and refusal polarities are pinned together in `bridge/tests/solana_value_path_holes.rs`. This closes the three arithmetic and authorization defects; it does not supply the missing mainnet snapshot/geyser feed or remove the oracle-custodial release path.

14.4 Outbound: settlement

A dregg proof settles on a foreign chain when that chain's verifier checks it natively. The settlement layer is a field-parameterized shrink (`circuit-prove/src/dregg_outer_config.rs`, generic over the STARK configuration): the recursion apex is re-committed with a hash native to the target chain's field, so the target's verifier hashes cheaply. The EVM is the BN254 instantiation: the apex passes through a gnark wrap to a Groth16 proof that a generated Solidity verifier checks on-chain, with forgeries rejected (`chain/gnark`, `chain/contracts/DreggSettlement.sol`). This path is deployed: a real proof of a dregg state transition verified on Base-Sepolia and advanced the settlement contract's proven root and height (`chain/DEPLOYMENTS.md`). The proof was a pre-generated fixture turn, and the Groth16 setup is a single-party development ceremony whose toxic waste is known to the operator; a production multi-party ceremony has not been run, and no mainnet deployment exists.

The Cosmos twin verifies the **same** BN254 proof — the same pinned statement, the same fixture — natively in a CosmWasm runtime: an `ark-bn254` port reproduces the two gnark pairing checks and advances the same proven-root/height pair (`cosmos-settlement/`). It accepts the real proof and rejects a forged root, proof point, or commitment under `cw-multi-test`, and it compiles to a deployable contract; it is not deployed to a live Cosmos chain, and the fuller IBC light-client path is named, not built. The Mina analogue — the Pasta instantiation of the same shrink —

is scoped, not built; the earlier Kimchi relay was removed because it never verified the proof in-circuit.

14.5 Per-chain maturity

chain	**inbound (prove holdings, then govern)**	**outbound (chain verifies a dregg proof)**
Solana	consensus-anchored holdings verify with a derived stake table; accept plus refusal polarities under test; the Lean core renders the weight verdict; fixture evidence only, no live feed	oracle-attested lock/mint runs; consensus-anchored lock verify built and fixture-tested, off-circuit; succinct wrapper named, not built; release oracle-custodial
EVM	ERC-20 storage-proof holding (EIP-1186) with secp256k1 binding, joined into governance	Groth16 verify on-chain; deployed to Base-Sepolia with a fixture proof on a development ceremony
Cosmos	bank-balance holding with secp256k1/bech32 binding	CosmWasm contract verifies the same BN254 proof in test; not deployed; IBC client named, not built
Robinhood Chain (EVM L2, id 46630)	tokenized-stock balance through the same EIP-1186 machinery against a supplied L2 root: structure-only, zero governance weight; the L1 rollup-anchor upgrade is named, not built	not demonstrated
Mina	—	scoped, not built (Pasta instantiation of the same shrink)

Table 8: Per-chain maturity of the two directions. Present-tense claims track this table: an entry says what verifies, under what evidence, and what remains named.

The Robinhood row illustrates the trust grading the whole surface uses. The proof machinery is identical to the Ethereum lane, but an Arbitrum-Orbit L2 carries no sync committee, so a holding verified against a supplied state root enters as structure-only and grants zero weight (`dregg-interchain-gov/tests/robinhood_inbound.rs`); the tier that grants weight requires deriving the root from consensus, and the Lean gate makes the distinction a theorem rather than a convention (`ProofOfHoldings.grantWeightCore_rpc_refuses`).

Nothing on this surface custodies real value today. The lock program has no recorded deployment and its tests sit in a workspace-excluded crate outside CI; the trustless vault verifier has accepted synthetic evidence but no mainnet snapshot/geyser feed; and release is oracle-custodial in both trust modes. The mechanism is built and adversarially tested. A production evidence source, ceremony, deployment, and external audit remain before it can hold value.

15 The units of the economy

The system has two economic units, and they are different objects. The **computron** is the internal metering unit: it prices execution, storage, and message delivery, and it is not a market asset. The **native token**, $DREGG, is an ordinary asset under the issuer discipline of the model — the worked instance of an asset whose issuer sits at the system's boundary. Neither unit adds a kernel mechanism. Every flow described below is a conserving move under the same value law that governs any other asset, and each role is stated with its maturity: *runs* (exercised green by a test or demo), *built* (real code, not exercised end-to-end live), *named* (a stated design without code). The canonical statement of record is `docs/TOKENOMICS.md`.

15.1 Computrons

A turn declares a fee, and the fee is the computron budget. The executor meters each action, effect, transfer, cell creation, proof verification, signature verification, and byte processed against a cost table, and it refuses a turn whose metered cost exceeds the declared fee (`turn/src/executor/execute.rs`). Fee coverage is checked against the agent's balance before execution begins. The cost table is a stated testing default, not a governed price list (`turn/src/executor/costs.rs`).

Fees are moves, not burns. A committed fee splits into conserving credits: half to the block proposer, three tenths to a federation treasury, and the remainder — at least one fifth, plus rounding dust and any share whose recipient is unconfigured — to a fee-well cell (`turn/src/executor/mod.rs`). The books therefore close. On the Lean executor, with mint and burn reshaped to issuer-moves, every state reachable from a value-empty genesis has per-asset total exactly zero: the issuer wells carry negative supply, and $\Sigma\delta = 0$ holds not merely per turn but identically along the whole reachable trajectory (`ReachableConservation.reachable_total_zero`). A configuration without a fee well burns the undelivered remainder; the deployed genesis configures the well.

Budgets distribute across execution silos without per-operation consensus. The mechanism is a Byzantine-tolerant bounded counter after Stingray [6]: an agent's balance splits into per-silo slices with ceiling $b(f+1)/(2f+1)$ for balance b and tolerated Byzantine silos f, silos debit locally against their slice, and rebalancing requires Ed25519-signed spending certificates with anti-replay debit digests, failing closed on a missing or invalid signature (`coord/src/budget.rs`). The counter is generic over any fungible resource; the computron carries no monetary semantics of its own. The same unit denominates storage quotas, relay inbox deposits, and relay-operator bonds. Supply today is devnet-shaped: a rate-limited faucet drains a pre-funded cell — a transfer, not a mint (`node/src/api.rs`).

15.2 The native token

$DREGG is a fixed-supply SPL token native to Solana, approximately 10^9 units, with no emission schedule, no inflation, and no protocol mint. The interchain posture reinforces the fixed supply rather than hedging it: dregg networks proofs, not tokens — other chains verify proofs *about* dregg state, and the token is never minted on another chain (`docs/deos/INTERCHAIN-MODEL.md`). Inside dregg the token appears only as a 1:1 mirror against units locked in a Solana vault. The bridge mint consumes a per-lock nullifier against the same committed nullifier set that note spends ride, so racing relayers cannot double-mint against one lock, and the executor refuses

any mint that would take live mirror supply above the committed locked amount (`turn/src/executor/bridge_ledger.rs`). Redemption burns the mirror before release. The repository pins no mainnet mint address in code; the binding is an operator environment decision.

15.3 Four roles

role	mechanism	maturity
services rail	metered service runs paid in stablecoin or token, dual-asset treasury	runs against mock chains; mainnet flip unfired
governance weight	non-custodial proof of holdings at a pinned slot	live local-validator feed; no mainnet snapshot feed
vault mirror	lock → mirror → burn → release	built; release oracle-custodial on every path
bond sink	none	deliberately absent

Table 9: The token's four roles, at current maturity.

The services rail. The payment backend sells metered service runs — AI narration on the deployed game surface — for either a stablecoin or the token: per-user derived deposit addresses, an idempotent per-payment credit ledger, and a dual-asset treasury in which the stablecoin funds real inference and fails closed when empty, while paid tokens accumulate in an illiquid operator treasury rather than being market-sold (`dregg-pay/`). Paying in the token earns a discount, set in basis points against an external price oracle. The rail is exercised end-to-end against mock chains; the mainnet go-live sequence is a written, unfired runbook (`docs/ops/PAYMENTS-GO-LIVE.md`), and no real token has yet been accepted for a service.

Governance weight. A holder proves that their own wallet held N units at a finalized snapshot slot, against a stake-weighted supermajority of Solana consensus anchored at a governance-pinned checkpoint, and receives vote weight N — no lock, no transfer, no wrapped token. A granted weight is backed by a consensus-proven holding at a finalized slot, and the grant leaves the on-chain state unchanged for every prior state (`ProofOfHoldings.weight_backed_and_noncustodial`); the weight verdict is rendered by the extracted Lean core with no Rust fallback (`dregg-governance/src/holding_weight.rs`). Weight is fixed per poll at the pinned slot, with a consume-once nullifier per (poll, holder, asset); the snapshot semantics are what defeat borrowed-balance weight and vote-sell-revote. The path runs in test, including rejection of a forged one-key stake table. A live `solana-test-validator` RPC feed has proven a real local SPL holding end to end; the mainnet snapshot/geyser source is unbuilt, so no real mainnet holding has been proven, and holding-weighted ballots land in a host-side ballot box rather than the verified vote engine.

The vault mirror. Spending the token inside dregg requires locking it into the Solana vault program (`solana-lock/`) and minting mirror units 1:1 against the observed lock, under the conservation gate above. The trust ladder is explicit. The production inbound slice is an M-of-N oracle attestation. A consensus-verified inbound successor — bank-state-derived stake tables, an authorized-voter-bound tally, a weak-subjectivity anchor — is built and green against fixture clusters (`bridge/tests/solana_lock_trustless.rs`) and has not verified real mainnet consensus. Release is oracle-custodial on every path; there is no trustless outbound. The reviewed defects in rooted finality, stake-set completeness, and rotation signer binding are closed and pinned

by `bridge/tests/solana_value_path_holes.rs`; the missing mainnet evidence source and the custodial release leg still keep real value off this path.

The absent bond sink. The bonded subsystems that exist are denominated in other units: relay-operator bonds and slashing run on computrons (`node/src/relay_dispute.rs`), and the launchpad's deployer-bond example is ETH-denominated (`tools/deployer-gate/`). The absence is an argued position rather than an omission: a conduct bond denominated in the token it polices loses value exactly when misconduct occurs, so bonds should be denominated in the quote asset (`docs/deos/FHEGG-CODEX-ROUND4.md`). A token-denominated collateral sink through the ordinary payment rail is named and unbuilt; any such design must price that correlated devaluation.

15.4 What does not exist

There is no staking yield. There is no burn mechanism: a slash seizure splits into a bounded restitution to the wronged party and a remainder moved to a configured treasury cell, both conserving transfers out of the bonded cell. No protocol fee routes to the token: the launchpad's only fee is a 30-basis-point pool swap fee that accrues to the pool's own reserves, and launchpad slashes compensate holders, never the platform. There is no play-to-earn; the deployed games' leaderboard reward carries no token (`docs/GAME-STRATEGY.md`). A reader applying the staking / burn / play-to-earn template will find each slot empty. The design is a fixed-supply asset whose demand is services, discounts, treasury accumulation, and governance weight, each at the maturity stated above.

15.5 The open decision: purchasing computrons

No peg, oracle, purchase path, or exchange rate between computrons and the token exists anywhere in the code; the token does not meter computation today. What exists are the docking points a purchase path would attach to. The relay fee policy carries a per-external-asset units-per-computron rate, designed for stablecoin deposit vouchers and disabled by default (`node/src/relay_service.rs`). The payment rail treats any bridged asset as spendable. The fee-distribution engine already routes every metered turn's value to named cells. Because the bounded counter is generic over any fungible resource, letting computrons be purchased in any asset at an operator- or market-set rate is a design decision above the kernel, not a kernel change — and it is deliberately open: the choice of which assets buy computation, and at what rate, is the one economic question the units leave unanswered. Until it is decided, the two units stay distinct: the computron meters, the token pays for services, and any flow ever built between them will be one more conserving move under the same value law.

16 The post-quantum floor

Two rows of the assumption floor (Section 17) break structurally under a quantum adversary: Shor's algorithm recovers ed25519 signing keys and computes the discrete logarithms that bind Pedersen value commitments. The remaining cryptographic rows face only generic search speedups. This section states what is proven about the system against a quantum adversary: a mechanized quantum oracle model, game reductions for the two lattice replacement primitives, a hybrid signature keystone, and finality safety under a disjunctive floor. The development is

connected end-to-end, from the adversary model to the deployed signing surface, with one named hypothesis remaining on the signature path.

16.1 The adversary model

The quantum adversary is modeled directly over Mathlib's finite-dimensional inner-product spaces: a state is a vector of EuclideanSpace $\mathbb{C}B$, a computation step is a norm-preserving linear bijection, and the random oracle is the basis permutation $|x, y\rangle \mapsto |x, y + H(x)\rangle$, a genuine unitary by construction (`QuantumOracle.oracleUnitary`). An adversary interleaves its own unitaries with q oracle calls. Over this model the one-way-to-hiding lemma bounds how well any such adversary distinguishes an oracle from a reprogrammed one: the semiclassical form gives $2\sqrt{q \cdot P_{\text{find}}}$ (`OneWayToHiding.o2h_bound`), and the double-sided form removes the $\sqrt{q}$ when the per-query difference vectors are pairwise orthogonal — the condition supplied by a deterministic, injective encryption whose reprogrammed point is efficiently recognisable (`DoubleSidedO2H.double_sided_o2h`). The orthogonality hypothesis is load-bearing rather than decorative: on an exhibited non-orthogonal pair the Pythagorean identity the proof rests on is false (`DoubleSidedO2H.orthogonality_load_bearing`).

The parameter-level payoff is a change of governing floor, not only a tighter constant. The tight KEM bound is $\min(\text{mlweBits}, \text{foCorrectnessBits} - \log_2 q) - 2$ (`DoubleSidedO2H.kemTightAdv_le`): the lattice term enters linearly, and the query budget survives only in the correctness term. At the recorded lattice estimate and a 2^{20}-query budget this is 152 bits, against 107 for the semiclassical bound (`DoubleSidedO2H.deployed_tightness_gain`), and it tracks the MLWE estimate — degrading the lattice floor degrades the bound, where the semiclassical figure would sit unchanged behind the message entropy.

16.2 The primitive games

ML-KEM (FIPS 203). IND-CCA security of the Fujisaki–Okamoto transform is grounded in MLWE together with the quantum random-oracle model (`MlKemIndCca.ml_kem_ind_cca_reduces_to_mlwe`). On the correctness side, the encaps and decaps cores are executable Lean definitions compiled to native code and called over FFI; the decode margin at the deployed modulus $q = 3329$ is a theorem with the flipping instance one past the bound exhibited (`Fips203Kem.decode_correct`), the encaps-to-decaps round trip is derived (`Fips203Kem.kyber_honest_roundtrip`), and the KEM round-trip hypothesis is discharged for the extracted cores (`Fips203Kem.extractedKemApi_fips203`) — no crate is trusted for it.

ML-DSA (FIPS 204). Unforgeability of the deployed scheme reduces to Module-SIS with the forking extraction proved rather than assumed: a forgery yields two self-target MSIS solutions on a shared commitment with distinct challenges (`ForkingDischarge.dregg_pq_is_eufcma_under_msis_discharged`, resting on `MSISHard` and a stated realizability bridge). Correctness holds at the real ML-DSA-65 dimension — the ring $R_q = \mathbb{Z}_q[X]/(X^{256} + 1)$ at $q = 8380417$, modules R_q^5 and R_q^6, an arbitrary matrix, an arbitrary Fiat–Shamir hash and challenge sampler, and an arbitrary secret (`Fips204FullDim.fullDimApi_fips204`). The residual there is the standard Dilithium rejection-termination fact: the seed carries the mask the rejection loop accepted, so the probabilistic termination of that loop is named as the boundary, not proved.

16.3 The hybrid keystone

Finalization votes carry an ed25519 **and** ML-DSA hybrid signature, and the composition is conjunctive: a certificate verifies only if both halves do. The keystone is that a total classical break buys the adversary nothing — with the classical verifier replaced by one that accepts every string, the hybrid remains unforgeable as long as the post-quantum half is (`HybridQuorum.hybrid_survives_classical_break`). The disjunction cuts both ways: a still-unforgeable classical half carries the hybrid through a future lattice break (`HybridQuorum.hybrid_survives_pq_break`). The hypotheses are discriminating — with both halves broken, a concrete forgery verifies (`HybridQuorum.both_broken_is_forgeable`).

16.4 Quantum-safe finality

The keystone reaches the ordering fabric (Section 6) at two levels. At the node, the vote collector admits a vote only if both signature halves verify, and the quorum counts only admitted votes; under a total classical break, an accepted quorum for a root still implies the honest committee signed it (`HybridFinalizationQuorum.hybrid_quorum_survives_classical_break`). At the protocol, no two conflicting blocks finalize at a height under $n > 3f$ together with the disjunction of the discrete-log and Module-SIS floors (`ConsensusSafety.consensus_safe_under_floor`); each honest member's unforgeability is produced from either floor by `HybridCombiner.hybrid_secure_if_either_floor`. A quantum adversary that breaks discrete log still faces MSIS, and the safety argument never consults the broken half.

16.5 The deployed surface and the one remaining hypothesis

The proofs above are about the shipped `dregg-pq` API, not a parallel model. The refinement relation states that the modeled signature scheme reads back exactly the API's public behavior — key derivation, context-separated signing, and the fail-closed Boolean verify — and it holds by construction (`DreggPqRefinement.dregg_pq_refines_sigscheme`), with a context-blind model exhibited as failing to refine. Domain separation is carried by unforgeability: a signature minted for one context does not verify under another (`DreggPqRefinement.dregg_pq_ctx_domain_separated`).

Exactly one FIPS hypothesis remains on the deployed signature path: `DreggPqRefinement.Fips204Correct` — for every seed, context, and message, `verify (keygen seed) ctx msg (sign seed ctx msg) = true` — instantiated at the API backed by the `fips204` crate. It says the crate implements the ML-DSA sign-to-verify round trip. It is load-bearing: without it, correctness of the deployed scheme is underivable (`DreggPqRefinement.badApi_not_correct`). Its specification-level counterpart is discharged at full dimension (`Fips204FullDim.fullDimApi_fips204`), so the open gap is crate-versus-spec — that the crate's byte-level behavior matches the modeled algorithm — rather than crate-versus-nothing. The KEM side carries no such hypothesis; its round trip is a theorem about the extracted cores.

16.6 What the development does not cover

The connected proof strengthens the floor; it does not eliminate it. Lattice hardness itself is assumed, not proved: MLWE and MSIS enter as hypotheses at stated boundaries and as recorded bit estimates in the parameter arithmetic, and every quantitative figure above is conditional on

those estimates. The adversary model is the query model over unitary oracles; it counts oracle calls and does not model side channels or implementation leakage. The rejection-loop termination of ML-DSA signing is a named probabilistic boundary. The extracted KEM and verify cores trust the `leanc`/FFI toolchain that compiles them, and their full-dimension byte codecs (the $n = 256$ ring interop) are named engineering residuals. Finally, the quantum posture of the rows the floor table marks "generic speedups only" — the hash, MAC, AEAD, and FRI carriers — is the recorded absence of a known structural attack, not a theorem of this development.

17 The assurance case

The system's guarantees are an artifact, not a narrative. `Dregg2/AssuranceCase.lean` states them **by guarantee**, assembles under each the keystone DAG that discharges it, and `#assert_axioms`-pins every name: the Lean build fails unless each theorem's full axiom set is exactly the kernel triple $\{\text{propext}, \text{Classical.choice}, \text{Quot.sound}\}$ — in particular, no `sorryAx`. `Dregg2/Claims.lean` is the corpus-wide per-keystone pin net behind it. The machine-readable form is the generated assurance catalog (`studio/assurance-catalog.generated.json`); the paper states the guarantees and defers the roster of record to it. Where a guarantee's substantive apex carries long module-local frame hypotheses, the file uses a heading anchor and pins the load-bearing keystone beneath it; the citations below name the keystone that carries the content, not the anchor.

17.1 The five guarantees and the running entry

A — Authority. *Every state change is justified by an unforgeable, non-amplified, fresh token chain.* The apex `AssuranceCase.authority_guarantee`: an introduction's conferred capability is a genuine non-amplifying subset of the held one (`EffectsAuthority.introduce_non_amplifying`) **and** the predicate discriminates — an amplifying grant is rejected (`EffectsAuthority.amplifying_grant_rejected`). Keystones include the per-mode admission soundness (`AuthModes.captp_sound`, `AuthModes.bearer_sound`, `AuthModes.token_sound`) and the dispatcher gate `AuthModes.captp_granted_le_held`. Floor: ed25519, HMAC, Poseidon2-CR.

B — Conservation. *Per asset, the resource sum is exactly zero across a turn and a run.* The apex `AssuranceCase.conservation_guarantee` over the genuine multi-asset ledger: the moved asset's total is invariant (`RecordKernel.recTransferBal_sum_conserve_moved`) and every other asset is pointwise untouched (`RecordKernel.recTransferBal_untouched`), lifted to the executor (`RecordKernel.recKExec_conserves`) and to the abstract monoid (`Spec.conservation_over_monoid`), with `Spec.committed_iff_cleartext` proving the committed and cleartext judgments agree. Floor: integer arithmetic only (Pedersen/DLog only when values are committed).

C — Integrity. *A receipt binds the whole post-state; a tampered input is rejected.* The load-bearing apex is the cross-bind `CommitmentCrossBind.runnable_binds_same_system_roots`, with the discriminating direction `CommitmentCrossBind.chC_bad_not_bridge` (a field-dropping commitment is not a faithful bridge) and the one-receipt welds (`Argus.Receipt.argus_commits_to_one_receipt`,

`Argus.Receipt.argus_circuit_executor_receipts_agree`). Floor: Poseidon2-permutation-CR — a second preimage is exactly the only way to forge a receipt for a different state.

D — Freshness. *No replay or double-spend; revocation takes effect at finality.* The apex `AssuranceCase.freshness_guarantee`: a committed spend's nullifier was fresh, is now present, and a repeat fails closed (`noteSpendStmt_no_double_spend`, `noteSpendStmt_replay_rejected`), with non-membership a witnessed opening (`NonMembership.nonmembership_sound` and `NonMembership.nonmembership_complete`), never a scan. Revocation is consensus-bound (`Liveness.revocation_needs_consensus`), with its undecidable dual pinned alongside (`Liveness.dead_undecidable`). Floor: Poseidon2-CR; PostGSTProgress for the at-finality leg.

E — Unfoolability. *A light client verifying a Q-chain learns A–D for the whole history while re-witnessing nothing.* The apex `AssuranceCase.unfoolability_guarantee` conjoins whole-history attestation (`RecursiveAggregation.light_client_verifies_whole_history`, Section 4) with whole-history conservation derived from the verification bit alone (`RecursiveAggregation.conserves_from_verification`), with the tamper-rejection direction pinned beside it (`RecursiveAggregation.tampered_aggregate_cannot_bind`, `RecursiveAggregation.leaf_pairing_defeats_swap`) and the strand realization (`Argus.Aggregate.argus_strand_light_client`). The whole attested history conserves along the fold (`RecursiveAggregation.attested_history_conserves`).

The guarantee also holds in game form. `LightClientUC.unfoolable_of_floor` reduces light-client soundness to exactly two floor carriers: STARK/Fiat-Shamir extractability (an accepting proof yields a satisfying execution witness) and commitment binding (a satisfying witness certifies the executor produced the state, itself sponge-CR). Under those two, no environment can make the client accept a state the executor did not produce. The contrapositive `LightClientUC.fooling_breaks_floor` extracts a concrete carrier break from any fooling attack: an environment that fools the client exhibits an accepting proof with no satisfying witness — a win against the extraction game itself. A working attack on the light client is therefore a working attack on a named floor row; there is no third option. Floor: FRI/STARK soundness (`EngineSound.recursive_sound`), Poseidon2-CR, ed25519, PostGSTProgress.

R — The running entry. *A, B, and C hold over what the node actually invokes.* The five guarantees above are stated over the kernel; guarantee R closes the gap to the deployment in one statement, `FullForestAuth.running_entry_sound`, about `FullForestAuth.execFullForestG` — the body behind the `dregg_exec_full_forest_auth` FFI export (Section 7): a committed gated forest conserves per asset, every delegation edge is non-amplifying (`FullForestAuth.execFullForestG_no_amplify`), and every node at every depth attests its gate (credential, caveats, capability-authority, and the per-asset obligation). The gate strengthens rejection without weakening the linear guarantees (`FullForestAuth.execFullForestG_unauthorized_fails`).

17.2 The assumption floor

Everything above is unconditional in the Lean-kernel sense **modulo** eight carriers — entering as `Prop`-portals (typeclass fields or hypotheses), never as axioms; nothing else is load-bearing anywhere in the case.

carrier	what it guards	quantum posture
Poseidon2-permutation CR	sponge / Merkle / state commitments reduce to permutation collision-resistance	generic speedups only
BLAKE3 CR	the out-of-circuit content / transcript hash	generic speedups only
ed25519 EUF-CMA	turn and strand-block signatures	**falls to Shor**
HMAC (PRF/MAC) unforgeability	macaroon caveat-chain tags	generic speedups only
AEAD confidentiality + integrity	sealed-value / disclosure payloads	generic speedups only
discrete-log hardness	Pedersen value commitments	**falls to Shor**
FRI / STARK soundness	a verifying proof attests its statement (`EngineSound.recursive_sound`); quantified below	generic speedups only
PostGSTProgress	eventual synchrony after GST — the consensus liveness carrier (Section 6)	not cryptographic

Table 10: The eight-carrier floor. The first seven are cryptographic; the last is liveness. Safety rests on the cryptographic seven; liveness additionally on the eighth. The quantum column records structural quantum attacks; "generic speedups only" means no attack better than Grover-class search is known.

The FRI/STARK row is the one carrier for which the repository records a bit calculation, because it is not at parity with the other six cryptographic rows and presenting it unqualified beside ed25519 EUF-CMA would suggest otherwise. The deployed recursion apex — the proof a light client actually checks — runs at extension degree 4 over BabyBear, log-blowup 6, 19 queries, and 16 grinding bits, with running tables floored at 2^{16} rows, so the initial evaluation domain is 2^{22}. Composing the batched-FRI theorem of Ben-Sasson–Carmon–Ishai–Kopparty–Saraf (2020) through the ethSTARK min rule yields a **57.98-bit density calculation** for that configuration; under the same authors' 2025 successor bound, whose exception count is linear rather than quadratic in the domain, the figure is **≈ 70.9 bits**, but the composition of that bound into a FRI soundness statement is not published — the calculation is this project's. Both figures bound the acceptance probability of a supplied proof and omit the DEEP/ALI terms, so they are optimistic arithmetic bounds on a density claim; extraction — that an accepting proof yields a witness — is the carrier itself, not a consequence of these numbers. The two-column structure of the bound is mechanized: `FriLedgerSound.query_ledger_does_not_determine_perFold` proves that the query ledger alone does not determine the per-fold proximity error, so the case never multiplies the two into a single headline. At the deployed extension degree the proven ceiling is 88.5 bits and 128 is unreachable at any query count; at extension degree 8 the mechanized ledger reads 122.60 bits on every shipped configuration (log-blowup 6, 36 queries, 16 grinding bits; `docs/reference/PROVEN-120-CONFIG.md`), with higher-query rows reaching 129.9, at the cost of rewriting the outer wrap and re-keying every verification key. The parameter space, both bounds, and the cutover levers are treated in full in Section 4.

Two rows fall to Shor's algorithm: a quantum adversary recovers ed25519 signing keys and breaks the binding of Pedersen commitments (hiding is unconditional, and conservation over cleartext integers never touches the row). The proven path off the brittle rows is hybrid. A hybrid ed25519 $\wedge$ ML-DSA certificate remains unforgeable when the classical half is replaced by an always-accepting verifier — a total classical break buys the adversary nothing against the AND-composition (`HybridQuorum.hybrid_survives_classical_break`) — and finalization safety survives a break of either half: `ConsensusSafety.consensus_safe_under_floor` derives that no two conflicting blocks finalize under $n > 3f$ together with the disjunction of the discrete-log and Module-SIS floors. The deployed `dregg-pq` signing surface refines the modeled scheme (`DreggPqRefinement.dregg_pq_refines_sigscheme`); on the signature path one hypothesis remains, named rather than laundered: that the `fips204` crate implements FIPS 204 (`DreggPqRefinement.Fips204Correct`). The post-quantum development — the quantum adversary model, the primitive games, and the hybrid reductions — has its own section; this table records only which rows it protects.

In particular: there is no trusted executor, no out-of-band "this was authorized" premise, and no field of the post-state left uncommitted.

18 Related work

Each stub names what dregg takes and where it differs.

Object capabilities [7], [8]. The authority model is Miller's: no ambient authority, *connectivity begets connectivity*, rights amplification via sealer/unsealers, the Granovetter introduction. dregg's contribution is to mechanize the **production** law — non-amplification and non-forgeability as two-valued, axiom-pinned theorems over the running executor (`EffectsAuthority.introduce_non_amplifying`, `EffectsAuthority.amplifying_grant_rejected`) — and to make every act of authority receipt-disclosed, hence verifiable after the fact by a third party.

Macaroons [9] **and Biscuits** [10]. Caveated bearer tokens with offline attenuation are the token lineage dregg inherits; HMAC caveat chains are on the assumption floor, and token adapters point inward to kernel capabilities. The difference is that dregg's caveat language is the same predicate algebra as cell programs and circuit obligations — an attenuation is checkable by the proof system, not only by the issuing service.

seL4 / l4v [11]. The methodological north star: a kernel whose specification, implementation, and proof live together, with explicit statements of what is and is not covered. dregg's analog of the refinement stack is the two-readings discipline (executor $\leftrightarrow$ circuit, welded per effect) plus the assurance case organized by guarantee; its analog of the l4v assumption statement is the eight-carrier floor. dregg verifies a distributed protocol substrate rather than a microkernel, and its executable artifact **is** the verified Lean rather than verified C.

Mina and recursive-SNARK light clients [12]. The aspiration "a chain you can verify on a phone" is shared, and the light-client theorem is the same shape (one root, recursive verification). dregg differs in what the proof witnesses: not only consensus-rule validity but per-step authority, conservation, integrity, and freshness — the proof attests the protocol's semantics, not just its block structure — and the verified statement is about the same Lean kernel the node executes.

Bridges, IBC, and zk light clients. Cross-chain systems answer "did this happen on the other chain?" in three ways: a committee attests (Wormhole, LayerZero, Hyperlane), each chain runs the counterparty's header verifier (IBC [13]), or a succinct proof of the counterparty's consensus replaces the header chain (zkBridge [14]). dregg takes IBC's discipline — the receiving side checks for itself — and the zk-bridge economy of checking succinctly, and removes the vote entirely: outbound, a target chain's contract verifies a Groth16 wrap of a dregg state transition directly (`DreggSettlement.sol` on the EVM, with a CosmWasm twin verifying the same proof); inbound, holdings on a foreign chain enter as consensus-anchored proofs while custody stays in the holder's wallet. What the proof witnesses differs as against Mina: per-turn semantics, not headers. The verifiers pass both polarities in test on a development trusted setup; none is deployed to a mainnet.

FRI soundness accounting [2]. Deployed STARK practice reports soundness from a parameter ledger: StarkWare's ethSTARK documentation [15] composes the commit-phase and query error terms as a minimum, with the commit-phase term proved by Ben-Sasson, Carmon, Ishai, Kopparty, and Saraf (BCIKS20 [16]) and improved from quadratic to linear in the domain by its 2025 successor (BCSS25 [17]). The field convention of quoting queries × blowup / 2 plus grinding as *the* soundness number keeps only the query column and silently drops the commit-phase term. dregg's ledger keeps the two columns separate as a theorem: `FriLedgerSound.query_ledger_does_not_determine_perFold` exhibits two deployed configurations with identical query ledgers and different per-fold postures, so neither column may stand in for the other. The same development transcribes BCIKS20′s commit-phase error so the reported bound under-approximates the paper's term; the assurance case states the resulting numbers for the deployed configuration.

Ceptre and linear-logic programming [18]. Reading state change as focused proof search in linear logic informs the substance discipline (value as a linear resource, verbs as structural rules). dregg fixes the dual orientation: search is untrusted and lives at the edges (solvers, intent matchers); the kernel only checks.

Verifiable game state and autonomous worlds. Dark Forest [19] demonstrated hidden-information play enforced by zero-knowledge proofs, and the MUD [20] line of on-chain game frameworks treats a game world as state whose rules the chain itself enforces. dregg takes the ambition — game state whose rules the operator cannot bend — and moves the guarantee's source: a game is a factory-minted cell whose moves are ordinary turns, so authority, conservation, and integrity arrive as inherited kernel theorems rather than per-title circuits, and a finished run verifies under the same light-client check as any other history. Hidden information is not a hand-authored circuit per mechanic but the per-viewer projection whose non-interference is proved once (the compositor stub below); a move a player could not see is absent from their view, not encrypted within it.

Blocklace / Cordial Miners [5], [6]. The ordering fabric: a signed DAG with equivocation exclusion, leaderless finality, and finality tiers as a lattice. dregg consumes it as the modal half of the step logic — when facts become common knowledge — and gates finality on the verified rule; its liveness enters the assurance case as the single PostGSTProgress carrier rather than diffusing through the proofs.

CapTP / OCapN [21]. The session layer for distributed object capabilities — sturdy refs, three-party handoff, promise pipelining — is the lineage of dregg's session surface. The kernel-

facing difference: a delivered handoff is admitted only through a verified non-amplification gate (`AuthModes.captp_granted_le_held`), and session machinery is kept out of the consensus-visible kernel (pipelining is turn composition; references are capabilities in slots). The sturdy reference is also the unit dregg's rehydratable surface ships: a persistable, attenuable handle to a witnessed scene, behind a membrane that re-checks authority at reacquisition.

seL4 capability description [22]. capDL writes a system's capability layout down once so a loader instantiates it and the deployment is checkable at the capability level. dregg's seL4 image places its cell-and-grant layout in the same relation — seL4 capabilities isolate the protection domains, dregg capabilities mediate the cells within — and grounds the attenuation leg of the inner discipline in a transcription of seL4′s own abstract specification, under one named bridge assumption (`Firmament.dregg_executor_cap_authority_grounded_in_seL4`).

Secure GUI servers [23]. The trusted-window lineage — a minimal GUI server that mediates input and labels output so domains cannot spoof or spy on each other — is where deos's compositor sits. A cross-domain compositor *trusts* its rendering process to provide isolation; deos *proves* per-viewer non-interference of the projection (`Deos.noninterference`, `Deos.hiddenCell_absent`) and the frame property of the compositing algebra (`Deos.render_frame_property`). A window is a capability, so the right to see is separated from the right to act by construction.

Hybrid post-quantum migration. Migration practice pairs a classical and a post-quantum scheme and accepts only when both verify — hybrid key establishment in TLS (X25519 with ML-KEM) [24] and dual-signature certificate proposals — on the argument that the pair is at least as strong as its stronger half. dregg takes the pattern and mechanizes the argument: a hybrid quorum certificate (an ed25519 vote quorum paired with FIPS 204 ML-DSA-65 signatures) is unforgeable if either half is (`HybridQuorum.hybrid_unforgeable_of_either`), and remains unforgeable under a total break of the classical half — the classical verifier replaced by an always-accepting one (`HybridQuorum.hybrid_survives_classical_break`). A compact variant replaces the per-signer ML-DSA concatenation with one committee-independent lattice threshold certificate whose unforgeability reduces to MSIS (`HermineHybrid.hermine_hybrid_unforgeable_of_either`). The hybrid path is implemented and staged behind a scheme flag; it is not wired into live consensus.

19 Limitations

Present-tense facts about the system as it stands. Each is a property a reader can check, not a roadmap.

The host-context seam. The verified Lean producer is the default state producer on the commit path, and its coverage is per effect kind. A turn that touches any effect kind outside the covered set is produced by the host Rust executor for that turn. For a further named set of effects the verified producer installs its post-state while its reconstituted root diverges from the Rust executor's reconstruction; the divergence is logged. The node reports the exact split — covered, uncovered, root-agreeing, and root-gap effect kinds — at `GET /api/node/producer` (`node/src/api.rs`). For turns the host arm produces, the host implementation is in the trust base, and the running-entry guarantee is stated over the verified entry, not over the host arms.

The proof-system carrier is explicit; its parameter ledger is a density evaluation, not a proof of adversarial soundness. The circuit layer's one recursion assumption is `RecursiveAggregation.EngineSound.recursive_sound`. Beneath it, the mechanized reduction takes `FriLdtExtractV3` — everything FRI delivers on an accepting run — as a hypothesis, and the Lean development contains no adversary or grinding model, so the figures that follow are evaluations of a published bound at the deployed parameters, not an extraction theorem or a bound against a prover strategy. At the recursion apex, the artifact a light client verifies (log-blowup 6, 19 queries, 16 grinding bits, tables floored at 2^{16} rows, hence a 2^{22} evaluation domain), the batched-FRI bound of Ben-Sasson, Carmon, Ishai, Kopparty, and Saraf (2020), composed by the ethSTARK minimum rule, evaluates to 57.98 bits; the 2025 successor bound by the same authors evaluates to about 70.9 at the same apex, but its composition into a FRI soundness statement is unpublished. The derivations and the pinned-source audit live in `docs/reference/FRI-BOTH-WIN-LEVERS.md`. The shipped parameter ledger (`FriKnobs`) carries no trace-height field, so it cannot express the commit-phase term ε_C that binds at the apex, where added queries and added grinding buy zero bits; that the query ledger does not determine the per-fold column is itself a theorem (`FriLedgerSound.query_ledger_does_not_determine_perFold`). At extension degree eight the mechanized ledger exceeds 120 bits on every shipped configuration (122.60 at log-blowup 6, 36 queries, 16 grinding bits; `docs/reference/PROVEN-120-CONFIG.md`). That configuration is not deployed; cutting over requires rewriting the degree-4 BN254 wrap and re-keying every verification key.

The post-quantum floor carries one FIPS hypothesis. The quantum-aware chain — a one-way-to-hiding adversary, through the lattice primitive games, to the hybrid keystone — is Lean-checked. The deployed ML-DSA scheme's correctness is conditioned on exactly one labeled hypothesis, `DreggPqRefinement.Fips204Correct`: that the `fips204` crate implements the sign-then-verify round trip. `DreggPqRefinement.dregg_pq_correct` derives correctness from that hypothesis and nothing else. A verified FIPS 204 implementation would discharge it; none is linked here.

Composition security is not machine-checked end-to-end in one system. The per-guarantee theorems are kernel-pinned, and the cross-corner welds (executor $\leftrightarrow$ circuit) are theorems. The core commitment-realization obligation is discharged in Isabelle/HOL — CryptHOL with the `Sigma_Commit_Crypto` Pedersen development (`Crypto/UCBridge.lean`, `uc-crypthol/Dregg2_FCom.thy`) — which widens the trust base to a second proof kernel and to a human-checked correspondence between the two systems' definitions. A universal-composability statement for the protocol stack as a whole is not a Lean artifact.

The global-zero value law does not yet cover the deployed genesis or legacy fees. The kernel's conservation guarantee is global: on every state reachable from a value-empty genesis, every asset's system-wide total — issuer wells included — is identically zero (`AssuranceCase.conservation_guarantee`, `Exec.ReachableConservation.reachable_total_zero`), and no verb moves any asset's sum (`TurnExecutorFull.ledgerDeltaAsset_eq_zero`). Two deployed paths sit outside the theorem's hypothesis and are not claimed by it: the devnet genesis seeds positive balances with no issuer well carrying negative supply (`node/src/genesis.rs`), and the legacy atomic-path fee epilogue debits a fee with no crediting move (`turn/src/executor/atomic.rs`). Until both are reshaped, the deployed chain's totals are offset by the genesis seed and decremented by legacy fees.

Guard expressibility has stated edges. The installable first-party atoms (`Exec.StateConstraint`, closed under the executor's Boolean algebra) are predicates over one

proposed transition: the old and the new state. Causal and temporal rules — predicates over the receipt trace rather than one step — are not installable atoms; a trace-shaped statement enters the grammar only through the witnessed branch, which defers it to the verify seam (`Spec.Guard.admits`).

Liveness is exactly as strong as its carrier. Safety guarantees are unconditional modulo the cryptographic floor. Finality and revocation-at-finality additionally rest on `PostGSTProgress`, eventual synchrony. A partitioned network stalls finality; it cannot forge it.

The embedded database executor runs the verified step on a reconstructed turn. In pg-dregg the in-backend producer runs the verified executor and takes its verdict and post-state as authoritative, but the extension does not link the turn codec, so it cannot decode a submitter's signed-turn envelope; it synthesizes a conserving wire turn instead, and the residual is stated at the seam (`pg-dregg/src/lean_producer.rs`). The range-attestation surface decodes the whole-chain proof transport in every build; the circuit verifier itself links only under the off-by-default `tier-c` feature, and the default build refuses rather than attesting (`pg-dregg/src/attest.rs`).

20 Conclusion

A turn is the exercise of an attenuable, proof-carrying token over owned state, leaving a verifiable receipt. From that one sentence the system follows: four substances under four disciplines, eight verbs as their structural rules, authority as constructive knowledge produced under a non-forgeability law, one predicate algebra at four polarities with two computed prices, and a receipt **Q** that is witness, projection, fold, and light-client claim at once.

The guarantees are not assertions about an idealized protocol. Their non-cryptographic cores are theorems about the Lean kernel the node runs, pinned to three axioms; their deployment force is conditional on the explicit eight-carrier floor and on the host-coverage seams in Section 19. Within that boundary, authority is non-amplifying, value conserves, receipts bind the post-state, spends are fresh, and one verified aggregate attests the whole history without re-execution (`RecursiveAggregation.light_client_verifies_whole_history`). The claim is therefore falsifiable at named boundaries: a fooling history must break the kernel-to-circuit correspondence, the proof-extraction carrier, or a commitment assumption.

The same capability and the same check carry across a distance parameter and down through the stack. A local microkernel object and a distributed cell are one attenuable reference whose single-machine limit is the strong case, not the degraded one; on seL4 the kernel's capability graph isolates the domains, dregg's mediates the cells within, and the verified executor itself runs inside its protection domain; in a database reads are SQL and writes are verified turns; and on the desktop a window is a capability whose rendering's non-interference and rehydration are kernel theorems. The same receipt underwrites an application's contract, a game's move, and a cross-chain settlement. The proof architecture holds all of this together as it evolves: the circuit rotates under proof, and every finalized turn stays verifiable across shapes, so the one root a stranger checks always covers the whole history.

The result is one check for a stranger, together with a precise account of what that check still assumes.

A Poseidon2 as garbling function for STARK-provable garbled circuits

This appendix details the garbled rung of the disclosure dial (Section 5): the two-party gate at which each party learns the verdict and nothing else. The Lean specification is `metatheory/Dregg2/Crypto/GarbledJoint.lean` — correctness (`GarbledJoint.garbled_correct`), input privacy (`GarbledJoint.garbled_input_private`), the joint-turn weld (`GarbledJoint.joint_turn_private_gate`), and the disclosure floor pinned to acceptance-only (`GarbledJoint.garbledDialFloor_is_bot`). The cryptographic realization is the garbling machinery in `circuit/src/garbled.rs`. Its proof path is the descriptor `GarbledEvalEmit.garbledEvalDesc` (`metatheory/Dregg2/Circuit/Emit/GarbledEvalEmit.lean`), authored in Lean and proved through `prove_vm_descriptor2`, with its correctness and mutation gate in `circuit-prove/tests/garbled_eval_emit_gate.rs`.

A.1 Motivation

Standard garbled-circuit constructions instantiate the garbling PRF with AES or SHA-256, primitives optimized for hardware execution but expensive inside arithmetic STARKs. Poseidon2 [3] is a permutation designed for efficient arithmetization over prime fields. Used as the garbling function in a Yao-style garbled circuit [25], it keeps the evaluation trace natively provable in a BabyBear STARK. The result is a verifiable garbled circuit [26] whose per-gate constraint cost is roughly three orders of magnitude below an AES-based equivalent (Section A.4).

A.2 The construction

A.2.1 Notation

Let $\mathbb{F}_p$ denote the BabyBear field ($p = 2^{31} - 2^{27} + 1$). A *label* is a vector $\boldsymbol{l} \in \mathbb{F}_p^8$: eight field elements, approximately 248 bits of entropy. For a wire w carrying a Boolean value $b \in \{0,1\}$, $\boldsymbol{l}_w^b$ is the label encoding value b on wire w. The function $\mathrm{P2} : \mathbb{F}_p^{16} \to \mathbb{F}_p^8$ is the first 8 output elements of a width-16 Poseidon2 permutation with S-box x^7 and 21 rounds (8 external, 13 internal), the parameterization pinned in `circuit/src/poseidon2.rs`.

A.2.2 Gate garbling

For a gate g with left input wire a, right input wire b, output wire c, and Boolean function $f_g : \{0,1\}^2 \to \{0,1\}$, the garbling key for input pair (i,j) is

$$\mathrm{key}_{i,j,g} = \mathrm{P2}\Big(\boldsymbol{l}_a^i \parallel \boldsymbol{l}_b^j[0..6] \parallel \big(\boldsymbol{l}_b^j[7] + g\big)\Big). \tag{1}$$

The 16-element permutation input packs the full left label, the first seven elements of the right label, and, in the final position, the right label's eighth element plus the gate index (`garbling_hash` in `circuit/src/garbled.rs`). Two distinct gates therefore feed the permutation distinct inputs except on a collision of the wire labels themselves. The garbled table consists of four ciphertexts,

$$\boldsymbol{C}_g^{i,j} = \boldsymbol{l}_c^{f_g(i,j)} + \mathrm{key}_{i,j,g} \quad \forall (i,j) \in \{0,1\}^2, \tag{2}$$

where $+$ is component-wise addition in $\mathbb{F}_p$. Evaluation recovers the output label by component-wise field subtraction:

$$\boldsymbol{C}_g^{i,j} - \mathrm{key}_{i,j,g} = \boldsymbol{l}_c^{f_g(i,j)}. \tag{3}$$

A.2.3 Encryption over the field

A garbled table entry is a one-time pad: each ciphertext masks an output label with a key used exactly once. Classical constructions take the pad over the XOR group $\{0,1\}^\lambda$; this construction takes it over the additive group of $\mathbb{F}_p^8$. Both are one-time pads over a group, and the hiding argument is the same: if the mask is indistinguishable from a uniform group element, adding it hides the label. Under the PRF assumption each of a gate's four keys is an independent pseudorandom vector — the four permutation inputs are distinct — so each ciphertext individually reveals nothing about the label it masks.

The field pad and the XOR pad differ in two respects, one gained and one given up. Gained: the field pad stays inside the arithmetization. Decryption, output = table entry − hash, is a single linear constraint per element in a BabyBear AIR, and it is exactly the decryption constraint the deployed descriptor enforces (`GarbledEvalEmit.decryption_body_zero_iff`). A bitwise XOR over the 32-bit integer representations would leave the field (the result can exceed p) and would cost a bit decomposition of roughly 31 binary constraints per element — the cost the choice of Poseidon2 is meant to avoid. Given up: the two-torsion structure of XOR ($x \oplus x = 0$), on which free-XOR optimizations rely (Section A.7). The construction samples every wire's two labels independently and correlates nothing across wires, so no algebraic relation spans two ciphertexts for the field structure to preserve — and no free-XOR-style optimization is available. Every gate carries four ciphertexts.

A.2.4 Point-and-permute

The garbled table rows are ordered by the least significant bit of the first element of each input label: $\pi(\boldsymbol{l}) = \boldsymbol{l}[0] \bmod 2$. The evaluator uses $(\pi(\boldsymbol{l}_a), \pi(\boldsymbol{l}_b))$ as a 2-bit index and performs one decryption instead of four trial decryptions. Label generation forces opposite color bits: the two labels of a wire are sampled independently, then the first element's low bit is cleared on the zero-label and set on the one-label (`random_label_pair`).

A.2.5 The comparison circuit

The deployed application is a private threshold comparison: the garbler holds a threshold t, the evaluator holds a value v, and the joint predicate is $v \geq t$ over 31-bit values. The circuit is an LSB-first borrow chain (`garble_comparison_circuit`): the borrow out of bit position i is a Boolean function of the borrow into it and the evaluator's bit v_i, with the garbler's known bit t_i wired into the gate's truth table. Each bit position is one two-input garbled gate, so a 31-bit comparison is 31 gates; the garbler's input occupies no gates at all. The comparison holds exactly when the final borrow is zero, so the zero-label of the last borrow wire is the "true" output label.

Two commitments bind the artifact. The circuit commitment is a domain-separated Poseidon2 sponge hash (`WideHash`: eight field elements, approximately 124-bit birthday collision resistance) over all garbled table entries; each output label is committed by the same hash. The evaluator's input labels are specified to arrive by oblivious transfer; the in-tree tests hand the evaluator its labels directly, and the repository contains no wire-level OT implementation.

A.3 Security argument

A.3.1 Garbling privacy

Privacy is argued in the simulation framework of Bellare, Hoang, and Rogaway [27].

> **Theorem (informal).** If Poseidon2 (width-16, x^7, 21 rounds over $\mathbb{F}_p$) truncated to its first eight outputs is a pseudorandom function of its 16-element input, then the construction above satisfies garbling privacy (prv.sim) with computational security $\lambda \geq 124$ bits at the garbling layer.

Proof sketch. The simulator, given only the circuit topology and the output labels of the true output, must produce garbled tables indistinguishable from real ones. Under the PRF assumption on P2:

1. For each gate g and each input pair (i, j) off the evaluation path, the ciphertext $\boldsymbol{C}_g^{i,j} = \boldsymbol{l}_c^{f_g(i,j)} + \mathrm{key}_{i,j,g}$ is indistinguishable from uniform: the key is pseudorandom, and adding a fixed value to a uniform element of $\mathbb{F}_p^8$ yields a uniform element (the group one-time pad of Section A.2.3).
2. For the single on-path row (i^*, j^*), the evaluator recovers $\boldsymbol{l}_c^{f_g(i^*,j^*)}$ and learns nothing about the opposite label, whose rows remain pseudorandom.
3. Gates are domain-separated through the key's final input element (the gate index added to a label element): two gates collide on a permutation input only if their wire labels collide, which the 248-bit label space makes negligible.

The 248-bit label space gives a birthday bound of 2^{124}: an adversary must evaluate P2 on the order of 2^{124} distinct inputs before observing a collision that could distinguish real garbled tables from simulated ones.

A.3.2 The mechanized privacy carrier

The Lean development states the privacy property over an abstract garbling kernel rather than the concrete permutation. `GarbledJoint.garbled_input_private` proves that the evaluator's transcript equals a simulator applied to the output bit alone; `GarbledJoint.garbled_input_private_indistinguishable` sharpens this to: two runs with the same outcome produce identical transcripts, whatever the private inputs. `GarbledJoint.joint_turn_private_gate` welds the kernel to the joint turn: a two-cell turn whose admission gate is a garbled predicate admits exactly when the joint condition holds, discloses only that one bit, and inherits the joint turn's atomicity unchanged. The floor of the disclosure dial for this rung is acceptance-only, formally the dial's bottom (`GarbledJoint.garbledDialFloor_is_bot`). These theorems model the protocol — garble, evaluate, transcript — not the STARK trace layout; the PRF assumption on Poseidon2 is the carrier connecting the model to the concrete construction.

A.3.3 What the STARK proves

The proof layer wraps evaluation in a STARK: the evaluator proves it performed the decryption chain correctly, so a third party can check the verdict without trusting the evaluator's report. The circuit statement is the 56-column descriptor `GarbledEvalEmit.garbledEvalDesc`, authored in Lean with its emitted wire string byte-pinned by a `#guard`, and proved through

`prove_vm_descriptor2`, the production descriptor prover. One trace row is one gate evaluation. The constraint families are:

- decryption correctness on every non-padding row — $\text{output}(i) = \overset{\text{table}}{\underset{\text{entry}}{}}(i) - \text{hash}(i)$ for each of the eight label elements, the field-subtraction decryption of Section A.2.3 (`GarbledEvalEmit.decryption_body_zero_iff`);
- booleanity of the six selector columns and exclusivity of the four gate-type flags;
- wire chaining across rows: where a row's chain flag is set, the next row's left label equals this row's output label (eight two-row window gates);
- first-row binding of the circuit commitment and output-label hash to the public inputs, and a first-row gate-index boundary (the every-row constraint families are additionally re-lowered as last-row boundaries, so no row escapes them).

The gate test `circuit-prove/tests/garbled_eval_emit_gate.rs` decodes the descriptor, checks it against an independently hand-built twin, proves an honest two-gate evaluation and re-verifies it, then refuses six mutated witnesses — a forged commitment input, a forged table ciphertext, a non-boolean selector, an ambiguous gate type, a broken wire chain, and a forged first-row boundary — each mutation biting a distinct constraint family.

Three scope facts bound what such a proof establishes; each is named in the artifact itself.

1. **The garbling hash is not constrained in-circuit.** The hash columns are free witness values; the digests — the per-gate P2 keys and both commitments — are computed by the witness generator in `circuit/src/garbled.rs`. The AIR proves the decryption algebra over those digests, and the digest-correctness binding is a named executor-verified carrier, stated as such in `GarbledEvalEmit.lean`.

2. **The commitments are bound in-circuit by a four-element prefix** of the eight-element digest. The full-digest equality is specified as a verifier-side check that has not been written; the repository prices the enforced prefix at roughly 62 bits.

3. **The path has no live consumer.** The descriptor, its gate test, and the garbling machinery exist and are exercised; no deployed surface currently issues or verifies garbled-evaluation proofs.

A.3.4 Composed security posture

The 124-bit figure above is a privacy bound at the garbling layer; it is not the security level of the composed artifact. A garbled-evaluation proof is checked by the same engine as every other proof in the system, and the engine's quantified posture bounds what acceptance is worth. At the deployed recursion apex (logarithmic blowup 6, 19 queries, 16 bits of proof-of-work grinding, tables floored at 2^{16} rows), the analysis of record evaluates the apex at 57.98 bits under the BCIKS20 accounting and about 71 bits under BCSS25 (`docs/reference/FRI-BOTH-WIN-LEVERS.md`). Those figures are analytic bounds on a supplied proof; the mechanized ledger carries no adversary or grinding model. Those figures do not establish an adversarial engine-soundness level, so they cannot be combined with the commitment and privacy figures as a minimum. What can be compared directly is narrower: the enforced commitment prefix is priced at roughly 62 bits and the garbling privacy argument at 124 bits, while the accepting STARK still relies on the explicit extraction carrier of Section 17. The commitment prefix and that carrier, not the garbling function, are the limiting obligations of the composed artifact.

A.4 Why Poseidon2 (not AES, not SHA-256)

The choice of garbling PRF is dictated by constraint cost inside a BabyBear STARK. The figures below are analytic estimates — constraint counts derived from each primitive's structure, not measured prover benchmarks. They price the design point at which the garbling hash is proved in-circuit; in the deployed descriptor the hash is an executor-verified carrier (Section A.3.3), so the estimates bound the cost of closing that seam and explain why the seam is closable at all.

AES-128. Each of the 160 S-boxes (10 rounds, 16 bytes) computes an inversion in $\mathrm{GF}(2^8)$; expressed over $\mathbb{F}_p$ this costs several hundred to a thousand multiplication constraints per S-box, on the order of 10^5 constraints per AES call.

SHA-256. The compression function performs roughly 1,100 32-bit AND, XOR, and rotate operations; each requires bit decomposition in a prime field. With gadgets that amortize decomposition across related operations, the cost is on the order of 2.5×10^4 constraints per call.

Poseidon2 (width-16, BabyBear). The permutation is native to the STARK: 141 S-boxes (8 external rounds of 16 lanes plus 13 internal rounds of one lane), each a single degree-7 constraint, with the linear layers constraint-free in the AIR — on the order of 1.5×10^2 constraints per permutation.

Garbling PRF	Constraints/call (est.)	31-gate circuit (est.)
AES-128	$\sim 100,000$	$\sim 3,100,000$
SHA-256	$\sim 25,000$	$\sim 775,000$
Poseidon2	~ 150	$\sim 4,650$

Table 11: Analytic constraint-cost estimates per garbled-gate decryption inside a BabyBear STARK. The 31-gate column is the deployed comparison circuit (one gate per bit of a 31-bit comparison). No entry is a measured benchmark.

The gap of roughly three orders of magnitude is the design's premise: with Poseidon2 the garbling hash is affordable in-circuit; with AES it is not.

A.5 Concrete parameters

Parameter	Value
Label entropy	$8 \times$ BabyBear ≈ 248 bits
Circuit topology	31 gates (LSB-first borrow chain, threshold wired into truth tables)
Garbled table size	31 gates $\times$ 4 rows $\times$ 32 bytes = 3,968 bytes
Commitments	`WideHash`: 8-element Poseidon2 sponge, $\sim$124-bit birthday collision resistance
STARK trace dimensions	32 rows $\times$ 56 columns
In-circuit public inputs	8 field elements (a 4-element prefix of each commitment)

Table 12: Parameters of the 31-bit comparison garbled circuit, derived from `circuit/src/garbled.rs` and `GarbledEvalEmit.lean` at the revision of record.

Proof size and timing are deliberately not restated. Earlier figures for this construction predate both the cutover to the descriptor prover and the current GPU prover, and no re-measurement accompanies this revision.

A.6 Limitations and assumptions

A.6.1 One-time evaluation

Each garbled circuit supports exactly one evaluation, the standard Yao limitation. A wire's labels encode a single Boolean value; reuse would let the evaluator learn both labels and decrypt all four rows of downstream gates. Repeated comparisons against the same threshold require freshly garbled circuits.

A.6.2 Evaluator learns the output

The evaluator necessarily learns the comparison result: the output labels ship with the circuit for decoding, and the evaluator identifies which one it recovered. Only the threshold, wired into the truth tables by the garbler, stays hidden from the evaluator. The STARK proof conveys the result to third parties without revealing anything else.

A.6.3 Semi-honest security

The construction provides security against semi-honest adversaries: both parties follow the protocol but may try to extract information from their views. Against a malicious garbler — one who garbles an incorrect circuit — standard techniques exist: cut-and-choose (garble κ circuits, open half for inspection, evaluate the rest) and dual execution (both parties garble and evaluate, then compare via an equality test, with one bit of leakage). Either composes with STARK verification, which proves correct evaluation of whichever circuit is selected. Neither is implemented in the repository.

A.6.4 PRF assumption on Poseidon2

Poseidon2 [3] is a 2023 construction. The permutation has been analyzed for collision resistance, preimage resistance, and algebraic attacks (Gröbner basis, interpolation, differential and linear), with claimed complexities above 2^{128} for this parameterization, but it has not received the decades of cryptanalysis applied to AES. A future break of Poseidon2′s PRF security would invalidate garbling privacy; it would not affect the integrity properties of the surrounding STARK.

A.7 Relation to prior work

Yao [25] introduced garbled circuits as the foundational technique for secure two-party computation. The construction here follows the standard Yao framework, differing in the garbling PRF and in the encryption group.

Bellare, Hoang, and Rogaway [27] formalized garbling-scheme security via simulation-based definitions. The security argument above targets prv.sim: the garbled circuit together with the output labels is simulatable from the circuit topology and the output alone.

Jawurek, Kerschbaum, and Orlandi [26] introduced verifiable garbled circuits, where the evaluator proves correct evaluation in zero knowledge, using Sigma protocols and cut-and-choose. Replacing that mechanism with a STARK yields a non-interactive, publicly verifiable, succinct proof of evaluation.

Grassi et al. [3] designed Poseidon2 as an arithmetization-friendly hash for use inside SNARKs and STARKs. The construction here extends its role from commitment and hashing to garbling: a PRF for encryption rather than a compression function for Merkle trees.

CAPSS [28] builds signatures from arithmetization-oriented permutations inside proof systems — the closest prior work in spirit: an AO primitive supplying cryptographic functionality (signatures there, garbling here) inside an arithmetic circuit.

Free XOR and half-gates (Kolesnikov and Schneider 2008; Zahur, Rosulek, and Evans 2015) reduce garbled-circuit size using the involutive structure of XOR over correlated labels (a global offset). The field-additive encryption used here samples labels independently and has no two-torsion, so these optimizations do not transfer; every gate carries four ciphertexts. For the 31-gate comparison circuit the absolute cost is small.

A.8 Security definitions

For completeness, the formal target.

Definition (prv.sim, adapted from [27]). A garbling scheme $\mathcal{G} = (\text{Garble}, \text{Eval}, \text{Decode})$ satisfies prv.sim if there exists a PPT simulator $\mathcal{S}$ such that for all PPT distinguishers $\mathcal{D}$, circuits C, and inputs x:

$$|\Pr[\mathcal{D}(\text{Garble}(C, x)) = 1] - \Pr[\mathcal{D}(\mathcal{S}(C, C(x))) = 1]| \leq \text{negl}(\lambda) \tag{4}$$

where $\text{Garble}(C, x)$ outputs the garbled circuit $\tilde{C}$ and input labels $\tilde{x}$, and $\mathcal{S}(C, C(x))$ receives only the circuit topology and the output value.

Claim. Under the assumption that $\text{P2} : \mathbb{F}_p^{16} \to \mathbb{F}_p^8$ (the first 8 outputs of the width-16 Poseidon2 permutation) is a (t, ε)-secure PRF for $t = 2^{124}$ and $\varepsilon = 2^{-124}$, the construction satisfies prv.sim with $\lambda = 124$ at the garbling layer.

The reduction is standard: the simulator replaces all non-output labels with fresh random values and computes ciphertexts as $\text{P2}(\text{random input}) + \text{random label}$; distinguishing real from simulated tables then requires distinguishing P2 outputs from uniform, contradicting the PRF assumption. The composed posture of a proof-carrying verdict remains as stated in Section A.3.3: the engine's quantified soundness at the deployed apex, not this bound, is the governing figure.

References

[1] Polygon Labs, "Plonky3: A Toolkit for Implementing Polynomial IOPs." [Online]. Available: https://github.com/Plonky3/Plonky3

[2] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Fast Reed-Solomon Interactive Oracle Proofs of Proximity," in *International Colloquium on Automata, Languages, and Programming (ICALP)*, 2018.

[3] L. Grassi, D. Khovratovich, R. Lüftenegger, C. Rechberger, M. Schofnegger, and R. Walch, "Poseidon2: A Faster Version of the Poseidon Hash Function," 2023, IACR ePrint 2023/323.

[4] A. Kothapalli, S. Setty, and I. Tzialla, "Nova: Recursive Zero-Knowledge Arguments from Folding Schemes," in *Annual International Cryptology Conference (CRYPTO)*, 2022.

[5] E. Shapiro and N. Sela, "Cordial Miners: Fast and Efficient Consensus for Every Eventuality," 2024, arXiv:2205.09174. [Online]. Available: https://arxiv.org/abs/2205.09174

[6] A. Auvolat, S. Beyer, and P. Kuznetsov, "Stingray: Bounded Counters for BFT Payment Channels," Jan. 2025, arXiv:2501.06531. [Online]. Available: https://arxiv.org/abs/2501.06531

[7] M. S. Miller, "Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control," in *PhD Thesis, Johns Hopkins University*, 2006.

[8] J. S. Shapiro, "The EROS Object System," in *ACM Symposium on Operating Systems Principles (SOSP)*, 1999.

[9] A. Birgisson, J. G. Politz, U. Erlingsson, A. Taly, M. Vrable, and M. Lentczner, "Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud," in *Network and Distributed System Security Symposium (NDSS)*, 2014.

[10] G. Couprie, "Biscuit: Authorization Tokens with Decentralized Verification." [Online]. Available: https://www.biscuitsec.org/

[11] G. Klein *et al.*, "seL4: Formal Verification of an OS Kernel," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[12] J. Bonneau, I. Meckler, V. Rao, and E. Shapiro, "Mina: Decentralized Cryptocurrency at Scale," 2020. [Online]. Available: https://minaprotocol.com/wp-content/uploads/technicalWhitepaper.pdf

[13] C. Goes, "The Interblockchain Communication Protocol: An Overview," 2020, arXiv:2006.15918. [Online]. Available: https://arxiv.org/abs/2006.15918

[14] T. Xie *et al.*, "zkBridge: Trustless Cross-chain Bridges Made Practical," in *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[15] StarkWare Team, "ethSTARK Documentation," 2021, IACR ePrint 2021/582. [Online]. Available: https://eprint.iacr.org/2021/582

[16] E. Ben-Sasson, D. Carmon, Y. Ishai, S. Kopparty, and S. Saraf, "Proximity Gaps for Reed-Solomon Codes," in *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2020.

[17] E. Ben-Sasson, D. Carmon, U. Haböck, S. Kopparty, and S. Saraf, "On Proximity Gaps for Reed-Solomon Codes," 2025, IACR ePrint 2025/2055.

[18] C. Martens, "Ceptre: A Language for Modeling Generative Interactive Systems," in *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2015.

[19] Dark Forest Team, "Dark Forest: A Decentralized Real-Time Strategy Game Built on Ethereum with zkSNARKs." [Online]. Available: https://zkga.me/

[20] Lattice, "MUD: A Framework for Onchain Applications and Autonomous Worlds." [Online]. Available: https://mud.dev/

[21] K. Varda, "Cap'n Proto: Capability-Based RPC Protocol." [Online]. Available: https://capnproto.org/

[22] I. Kuz, Y. Liu, I. Gorton, and G. Heiser, "capDL: A Language for Describing Capability-Based Systems," in *ACM Asia-Pacific Workshop on Systems (APSys)*, 2010.

[23] N. Feske and C. Helmuth, "A Nitpicker's Guide to a Minimal-Complexity Secure GUI," in *Annual Computer Security Applications Conference (ACSAC)*, 2005.

[24] K. Kwiatkowski, P. Kampanakis, B. Westerbaan, and D. Stebila, "Post-quantum Hybrid ECDHE-MLKEM Key Agreement for TLSv1.3," IETF Internet-Draft draft-ietf-tls-ecdhe-mlkem-05, May 2026. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-tls-ecdhe-mlkem/

[25] A. C.-C. Yao, "How to Generate and Exchange Secrets," in *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1986.

[26] M. Jawurek, F. Kerschbaum, and C. Orlandi, "Zero-Knowledge Using Garbled Circuits: How To Prove Non-Algebraic Statements Efficiently," in *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[27] M. Bellare, V. T. Hoang, and P. Rogaway, "Foundations of Garbled Circuits," in *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[28] A. Aly, T. Ashur, D. Kales, E. Orsini, and M. Schofnegger, "CAPSS: Signatures from Arithmetization-Oriented Permutations," 2024, IACR ePrint 2024/252.