# Molt Petit: A Machine-Checked Slot-Based BFT Consensus Protocol with Recursive Certificates

George Agapov · Yak Software, Andorra
September 2026, draft · yak.finance/molt-petit

yak.

**Abstract**

Molt Petit is a slot-based Byzantine-fault-tolerant consensus protocol whose entire light-client contract fits in one sentence: hold the genesis and a clock, fetch a constant-size recursive certificate from anyone, check that it verifies and that its tip is recent, and act on everything $n$ blocks deep. Validity is a *density* rule — every matured $n$-slot window must carry blocks from two thirds of the fixed roster — with no votes and deterministic, fixed-depth finality. All safety claims are machine-checked in Lean 4 — the core claims about the imported Rust validator source itself, proved sound against the model and re-used as the recursive-proof circuit, and today's key-rotation results at model level. Beyond the classical counting argument, we prove that forged chains advance at half real-time speed — which makes the verifier's recency check a load-bearing safety ingredient — and we develop key rotation inside consensus against an adversary who steals keys and keeps them forever: three rotation modes with exactly stated operator and client duties. Measured with post-quantum signatures on commodity hardware, the verified consensus rule adds under 2% to any certificate's proving cost.

## 1 Introduction

Consider a light client that is offline most of the time — a wallet, an embedded device, a bridge contract — and occasionally needs to act on the current state of a small consensus system. Today it either replays history, trusts a service, or verifies a chain of committee hand-offs. Molt Petit is a consensus protocol built so that this client can need exactly three things: the deployment's genesis, a clock, and a constant-size *certificate* it can fetch from anyone. If the certificate verifies and its tip is recent, the client may act on everything $n$ blocks deep — and that guarantee is a machine-checked theorem about the shipped validator code, not a design argument.

The protocol is deliberately small. A fixed roster of $n$ participants produces blocks in public round-robin slots. There are no votes, no view changes, and no fork-choice metric in the safety argument — fork choice is the trivial highest-valid-tip rule, load-bearing only for liveness; the one structural rule is *density* — a chain is valid only if every matured window of $n$ slots contains blocks from at least two thirds of the roster (Section 3). Density makes deep forks impossible while the fault budget holds, which turns finality into a fixed depth rather than a probability. What a node holds is never history but a recursive proof of the validity rule itself, plus a short suffix of signed blocks (Section 4).

Two ideas carry the security argument beyond the standard counting. First, *recency is a safety ingredient, not a freshness nicety*: a per-slot adversary — one that corrupts producers slot by slot — can slowly harvest coerced signatures into a perfectly valid-looking fork, but each block's id commits to its parent's signature, so forged suffixes provably advance at half real-time speed — any fork old enough to be dangerous is too stale to be accepted (Section 6.2). Second, *key rotation lives inside consensus*: every block declares its signing-key version, so rotation is ordered by the chain like any other event. Against an adversary who steals keys outright and keeps them forever, the paper gives three rotation modes with exactly stated operator and client duties — down to a theorem that fixes, in the one mode built for actively syncing clients, the single rule such a client must obey (sync at least once every $n$ slots, Section 6.3); clients of the other two modes obey no rule at all.

The engineering claim is as important as the theorems. The validator is one Rust file; it is *imported* into Lean 4 (via Charon + Aeneas) and proved sound against the model, so the core safety theorems are about the code that runs (the rotation pins are checked at model level today, Section 7); the same source is re-used as the recursive-proof circuit, so the prover proves the predicate that was verified; and a separate TypeScript implementation is proved sound against that *same* Lean model — which is what keeps the model implementation-agnostic while it is developed by agentic coding. A build-enforced audit fixes what every headline theorem may assume: the axioms of classical logic, and nothing else. Measured on commodity hardware with post-quantum signatures, the verified consensus rule is under 2% of any certificate's proving cost (Section 7) — formal consensus is effectively free on top of the application.

The cost of this simplicity is stated up front and in Section 9: a closed, equally weighted roster; no liveness recovery (a starved window halts the chain by design); and safety that depends on correct clocks rather than holding under arbitrary network timing. Where those trades are acceptable — permissioned settings, bridges, embedded fleets — the result is a consensus layer whose entire client contract fits in one sentence and whose proofs re-verify from a fresh checkout. The next section states that contract completely, mode by mode, before any definitions.

# 2 How to run it: the operational model

One protocol runs throughout this paper. A fixed roster of $n$ participants produces blocks in public round-robin slots; every block declares the signing-key version it was signed under; a node holds a constant-size certificate plus a short suffix of recent signed blocks; and a verifier accepts a chain only if it validates and its tip is recent — at most $n$ slots old on the verifier's own clock — then trusts everything $n$ blocks deep. Key rotation is part of the protocol from the start: a deployment completes the validator by fixing one of three rotation *pins*. A pin bounds the key version a block may declare — a floor in modes 1–2, exact per window of a fixed grid in mode 3 — and so decides when a retired key becomes dead; that choice is the mode (1–3 below). Mechanically the modes differ only in what computes the pin; in what they guarantee, they differ only against key *theft*. Producers rotate; clients never react to rotations — each client duty below is a fixed rule in time, never a response to an event. (Names in `typewriter` are machine-checked Lean theorems; every one resolves in the companion library `Molt/`.)

**Mode 0: any pin, when keys can be lost but not stolen.** Every adversary in this paper can make a producer *lose* its key — destroy it, wedge its signer, force a rotation. Against an adversary with only that power, all three modes give the same guarantees to the same client: **hold the genesis and a clock; sleep as long as you like**. Loss needs no special safety machinery. The loser recovers by its mode's own discipline — an in-band rotation in mode 1, a later scheduled generation in mode 2, the next lockstep switch in mode 3 — at the price of its missed slots (operational discipline; mode 1's half is also checked in the core, `liveness_produce_blockK`). And in every mode the corruption the theorems count shrinks back to the plain controlled-slot budget of Section 5 (Lean `badKeyrot_lossOnly`, `badSched_lossOnly`, `exposedSched_lossOnly`). A deployment that rules out theft by other means may run any mode and read no further than light-client safety and the forged-time bound (Sections 6.1–6.2).

**Mode 1: reactive rotation — for clients that stay current.** Producers rotate whenever they choose, routinely or on suspicion, and *no one knows in advance when or how rotations will unfold* — there is no schedule, public or private. The client's whole duty: **re-verify a chain at least once every $n$ slots** — the same $n$ as the recency rule — keeping as its anchor the block $n$ deep below the last verified tip, and refusing chains that do not contain it. A client that misses its cadence must stop acting and re-join from a fresh trusted checkpoint, exactly as if it were new — operational discipline the theorems force rather than state: past the cadence the theorems' fault-budget assumption becomes one no deployment can stand behind — quiet thefts accumulate until nothing satisfies it (`no_budget_beyond`), and re-joining is a fresh trust event outside the formal development. The deployment stays safe for as long as rented slots (controlled for one slot, then lost) plus stolen producers stay within the fault budget on every window starting in the trailing $5n$ slots or later — *with thefts counted as totals for the whole stretch: rotating a stolen key away caps the damage, it does not refund the budget* (Theorem 3).

**Mode 2: scheduled rotation — for clients that sleep.** The key version in force at any slot is a *public function of the slot*, fixed and published before the deployment starts: every producer, verifier, and client knows the entire rotation schedule in advance, and the validator refuses versions below it. Every mode signs under keys delegated from one offline master key (a *cold root*) per seat (Section 3); what mode 2 adds is *when*: the operator derives each period's signing key — each *generation* — just before its scheduled span (its *era*) begins, the named operational assumptions *not-before* and *cold-root custody* (Section 6.3). The client holds the genesis, a clock, and the constant that defines the schedule; it may stay offline arbitrarily long and sync whenever it wakes. No anchor, no cadence, no erasure.

**Mode 3: free-cadence lockstep — for clients that sleep and operators that erase.** No schedule anywhere, not even a constant. The roster advances one shared generation counter at moments of its own choosing; validity forbids mixing generations within any window of the fixed window grid —

| symbol | meaning | where |
|---|---|---|
| $n$ | roster size; also the density window (slots), the recency bar (slots), and the trust depth (blocks) | §3 |
| $q = \lceil 2n/3 \rceil$ | window quorum | §3 |
| $f = \lfloor (n-1)/3 \rfloor$ | per-window fault budget | Asm. 1 |
| $\tau$ | slot duration (seconds) | §7 |
| $now$ | verifier's clock; recency test $now \leq tip.slot + n$ | §3 |
| $H(\cdot)$ | the block-id hash | §3 |
| $keyFor(i, j)$, `keyIndex` | seat $i$'s public key at version $j$; the version a block declares | §3 |
| $\rho$, $T$ | per-window rented slots and stolen-key producers, $\rho + T \leq f$ | §6 |
| $R$, $gen(s) = \lfloor s/R \rfloor$ | rotation-schedule period; scheduled key version (mode 2) | §6 |

Table 1: Notation. Every timing rule in the paper is set by $n$: the density window, the recency bar, the trust depth, the confirmation depth at which a mode-1 rotation takes force, and the mode-1 client's sync cadence (Section 6). There is no other timing constant.

slots $[Wn, Wn{+}n)$ — and never lets the counter decrease across grid windows; and retired key material is physically destroyed at each switch (hardware modules or RAM-only keys — storage that does not truly erase does not qualify). The client holds the genesis and a clock — even less than mode 2 — and may sleep arbitrarily long. The price sits entirely with the operators: lockstep coordination, and erasure that is real. (Mode 3's guarantees are currently proved for full chains, and erasure's per-generation credit against the budget is designed but not yet formalized — today's theorems count thefts across all generations within one budget, with or without erasure; both are future work, Section 6.3.)

Everything that follows makes these paragraphs precise: the protocol and its validator (Sections 3–4), what a deployment must assume (Section 5), and the theorems each contract rests on (Section 6).

# 3 The protocol

Machine-checked claims cite their Lean names in `typewriter`; every cited name resolves in the companion library `Molt/`, except the per-implementation bridge names of Section 7, which live with their pipelines.

## 3.1 Setting

Time is divided into slots of a fixed duration $\tau$. A fixed, closed set of $n$ participants — the *roster* — produces blocks. Slot $s$ belongs to roster seat $s \bmod n$ (`producer`): who may produce, and when, is deterministic and public; there is no leader election. Two constants derived from $n$ appear throughout: the *quorum* $q = \lceil 2n/3 \rceil$ and the *fault budget* $f = \lfloor (n-1)/3 \rfloor$ — the number of slots per $n$-slot window that faulty participants may control (Assumption 1 makes this precise). Table 1 collects the paper's symbols.

## 3.2 Blocks and chains

A block is six fields:

$$B = (slot,\ height,\ prev,\ id,\ contentsHash,\ keyIndex).$$

The payload (transactions, in a deployment) travels out of band; `contentsHash` commits to it, and no consensus rule ever looks inside it. `keyIndex` is the version of the signing key the producer used: key rotation is part of the consensus state, carried in band, and Section 6 builds on it. Every block travels with its producer's signature over it — carried alongside the six fields, never inside them. `prev` is the parent's id, and `id` is a hash of the block itself, computed and checked by the deployment's hashing layer under a fixed contract:

$$id \ = \ H(slot,\ height,\ prev,\ parentSig,\ contentsHash,\ keyIndex).$$

The consensus rules treat ids as opaque. Two preimage choices do real work later. The *parent's signature* is inside the hash: that is what forces anyone fabricating a chain to produce its signatures one after another rather than all at once (Section 6 turns this into a wall-clock bound). And the *key version* is

inside the hash, so two blocks differing only in it hash differently — without that, the hash assumption becomes unsatisfiable the moment any producer has rotated.

A chain is a sequence of blocks with heights increasing by one, slots strictly increasing, and each block linking to its parent's id, rooted at a genesis block — height zero, no parent (`linksOk`, `genesisOk`); that the root is the deployment's genesis is what the theorems hypothesize (Section 5).

### 3.3 The density rule

There are no votes and no longest-chain comparison. A chain must be *dense* (`validChain` names the combined structure-and-density checks): every matured window of $n$ consecutive slots must contain at least $q$ blocks of the chain, where a window $[u, u+n)$ has matured once the tip has reached its end, $u + n \leq tip.slot + 1$ (`denseSoFar`). Read it operationally: a chain is acceptable evidence only if, at every point of its history, at least two thirds of the roster was actively extending it.

Density is the whole safety mechanism. If two chains diverge and both stay dense, each must collect $q$ blocks in any window past the divergence; since $2q > n + f$, some slot in that window carries a block of *both* chains and belongs to an honest producer — and an honest producer extends one chain per slot, never two. Every safety theorem in Section 6 is this counting argument made precise and machine-checked.

Chain validity has one more condition, riding on the `keyIndex` field: along the chain, a producer's declared key version never decreases (`keyMonoOk`) — rotation only ever moves forward, and the chain's order is the agreed order in which rotations happen. Structure, density, and this monotone rule together are `validChainK`: *the* chain validity of this paper. There is exactly one protocol here, with key rotation built in from the start; a deployment completes it by fixing one of three rotation *pins* (Section 6.3).

### 3.4 What a node holds

A node does not hold history. It holds a *certified chain*: a constant-size certificate plus a short suffix of recent signed blocks. The certificate attests everything below the suffix and exposes a *claim*: the tip slot, height, and id it vouches for, plus a small boundary buffer of near-tip blocks so that window density can be counted across the seam. In the flagship implementation the certificate is a *recursive* zero-knowledge proof: each new proof checks the previous proof plus the newly folded blocks, so one constant-size object ends up attesting arbitrarily long history.

The rules touch cryptography only through two small interfaces that a deployment implements:

- `SigOps` — *sign* and *verify* — together with the versioned key map $keyFor(i, j)$, the public key of seat $i$'s signing key at version $j$ (a separate parameter, `KeyRegistry`, in the model; the implementation's dictionary folds it into `SigOps`);
- `CertOps` — verify a certificate, read its claim, and extend it by one validated block; the implementation's dictionary (Section 7) also exposes a by-tip-id lookup of locally held certificates, which compaction uses.

One deployment note on *keyFor*, because it is where a trusted directory service would creep in: there is none, and a verifier looks nothing up. *keyFor* is a mathematical parameter of the model — which public key each (seat, version) pair denotes — and a block carries its own witness for it. A block's "signature" is in practice a bundle — the signing public key, a delegation certificate binding that key to seat $i$ *and version* $j$, and the signature proper — verified against a fixed set of *cold root* keys, one per roster seat, created at the deployment ceremony. The delegation certificate must bind the version: one delegate key answering for several versions is exactly the key reuse that Section 6 has to exclude by assumption.

### 3.5 The verifier rule

Everything a consumer of chains — above all, a light client — does is one rule. Accept a certified chain if and only if:

1. it *validates*: the five checks of Section 4 — certificate, signatures at declared versions, linkage, density, and the key-version discipline with the deployment's pin (`validCertifiedChain` plus the mode's key checks);
2. it is *recent*: $now \leq tip.slot + n$ on the verifier's own clock. (This clock comparison is the one line a deployment implements outside the verified validator — Section 9.)

It then trusts the blocks $n$ deep below the tip. Both halves are necessary: validity without recency admits carefully aged forgeries (Section 6 quantifies exactly how old a forgery must be), and recency without validity means nothing. The same $n$ serves as density window, recency bar, and trust depth — Section 6 shows why one size fits all three roles. At the prototype's parameters ($n = 7$, $\tau \approx 4\,\mathrm{s}$) the rule reads: accept only chains whose tip is at most $\approx 28\,\mathrm{s}$ old, then trust everything from about a minute back.

# 4 The rules a node runs

A node's whole life is three operations on its certified chain. All three are pure functions — deterministic, no I/O, no hidden state — which is what lets one source of truth serve as reference implementation, Lean model, and circuit statement at once (Section 7).

**Validate** (`validCertifiedChain`). Accept a certified chain only if all of the following hold:

1. the certificate verifies (`CertOps.verify` — one cheap check, e.g. of a recursive proof);
2. every suffix block's signature verifies under *keyFor*(slot's producer, declared version);
3. the first suffix block extends the claim's tip: height one up, slot strictly later, `prev` equal to the tip id — and every adjacent suffix pair links the same way;
4. every window that matures within the suffix is dense, counted over the claim's boundary buffer plus the suffix (`validSuffix`);
5. the key-version discipline holds: a producer's declared versions never decrease (`keyMonoOk`, Section 3), and every block's declared version satisfies the deployment's rotation *pin* (a floor in modes 1–2, exact per grid window in mode 3) — the one mode-specific check, fixed at deployment. These two are stated over the full-chain form of the validator (`validSignedChainK'`; the same validator under the mode 2–3 pins); at certificate level they are proved for modes 1–2 (`keyrot_recent_certified_suffix_agreement`, `sched_recent_certified_suffix_agreement`) and future work for mode 3 (see the "honest scope" discussion in Section 6.3).

The model replays the structural checks from genesis (`validChain`); a deployed node checks only the windows that mature with each arriving block. The two are equivalent — earlier windows were checked when the parent was validated — so the incremental check is an optimization, not a loosening.

**Produce** (`produceBlock?`). Only in the caller's own slot: build the next block on the current tip, sign it, append it to the suffix — and ship the result only if it passes the same `validSuffix` every other node runs. That final gate is the point: nothing on the production path has to be trusted for safety, because anything malformed it could emit is exactly what every validator rejects.

**Select** (`selectChain`). Adopt a candidate chain only if it validates and its tip is strictly higher than the current one's. Selection never carries safety: the theorems of Section 6 hold for *any* two chains a verifier accepts, however they were selected. Selection matters only for liveness.

**Compaction.** Periodically a node folds the oldest suffix blocks into the certificate and keeps the suffix short. Certificates carry no authority — they are proofs — so any node may run the prover, and one untrusted proving service suffices; if proving lags, the suffixes simply grow until it catches up, and validity is unaffected. Because certificates arrive in batches, compaction scans the possible cut points from newest to oldest and jumps at the first position for which it holds a verified certificate whose claim equals the current claim advanced by exactly the dropped blocks. The compacted chain must then pass `validCertifiedChain` like anything else; compaction, like production, is outside the trusted base.

# 5 What a deployment must provide

Every assumption below except the clock appears literally as a hypothesis of the theorem that cites it — Assumptions 1–4 in Theorems 1–2, Assumption 6 in the liveness theorem; the rotation results replace Assumptions 1–2 with rescoped forms defined in Section 6.3's adversary paragraph. Lean's axiom audit confirms that nothing further is assumed behind the scenes. The clock is the one exception by nature: the theorems take the verifier's reading *now* as a parameter, and Assumption 5 says what that parameter must track in the real world.

The adversary, stated once: message delivery is entirely adversarial for every safety result (timeliness enters only for liveness, Assumption 6); corruption is dynamic and per-slot, within the budget below, with full control of a bad slot's producer, including the power to make it sign arbitrary messages; the verifier's clock may err by up to $\approx n/2$ slots while the adversary's timing is unconstrained. What the adversary cannot do is exactly Assumptions 2–4: forge signatures of keys it does not control, find hash collisions among blocks that occur, or forge certificates.

**Assumption 1** (Fault budget)**.** *In every window of $n$ consecutive slots, at most $f = \lfloor (n-1)/3 \rfloor$ slots are* bad*: during a bad slot the adversary fully controls that slot's producer, including feeding it arbitrary content to sign. Corruption is per* slot*, not per participant — the same participant may be controlled in one of its slots and honest in the next. (Lean:* `FaultBounded`*.)*

**Assumption 2** (Signatures)**.** *The model gives each participant a* signing log*: at most one block per slot, the one its honest signing path signed (*`SigningLog`*). The assumption bundles:* (a) custody *— honest nodes sign only through that path, and their secret keys never leave them;* (b) unforgeability *— no one can produce a valid signature on a message the key holder never signed, even after obtaining signatures on messages of their own choosing (the standard EUF-CMA notion); here the attacker gets exactly that power at every bad slot, whose producer can be made to sign anything; and* (c) *the small leftover statement — the formal* residue *— that the theorems actually use: for any block of a valid chain* whose tip is recent*, produced in an honest slot and carrying a verifying signature, that block is the unique entry of its producer's signing log for that slot. (Lean:* `SigUnforgeableRecent`*.)*

The recency scoping in (c) is essential, not a convenience: an adversary can walk away from a bad slot holding coerced signatures stamped for that producer's *future* slots, so no real deployment could ever satisfy the unscoped statement. What makes the scoping sound is the forged-time bound (Section 6): a chain old enough to have been assembled from harvested signatures is precisely a chain the recency rule already refuses. Section 6.2 shows that (c) is derived, not primitive.

One forward note for key rotation: with versioned keys, custody and unforgeability must hold for every version a verifier can still be asked to check — as stated, a single leaked *retired* key falsifies the assumption. Section 6 replaces it, for the rotation results, with an explicitly scoped assumption — the *surface* defined there — in which stolen keys of any version sign anything forever and safety is recovered by validator rules.

**Assumption 3** (Hash collision resistance, over blocks that occur)**.** *Any two blocks that each either equal the genesis or carry a verifying producer signature, and that have equal ids, are equal. This is classical collision resistance in the form a proof can consume, not a weakening of it: plain injectivity of $H$ is false outright — ids are shorter than blocks, so colliding values exist in the type — while collision resistance says only that no adversary ever* exhibits *one. Scoping the domain to blocks that can actually occur is what turns the second statement into the first; every block of a validated certified chain is in that domain. We call this* hash injectivity*; under rotation the domain is version-stamped signed blocks — for the chain-level sync-rule results (Theorem 3), blocks verifying under* some *registered version of their producer (Lean* `KeyStealingSigned`*); for the scheduled, lockstep, and certified results it narrows to blocks verifying under their* declared *version (*`SignedDeclared`*), a strictly weaker assumption — Section 3 explains why* `keyIndex` *is in the preimage. (Lean:* `SignedHashInjective`*.)*

**Assumption 4** (Certificate grounding)**.** *A certificate that verifies carries a claim built from the deployment genesis by folding in one block at a time, where each fold checks the link rules, the density of newly matured windows, and a verifying producer signature on the folded block. Deliberately minimal: it does* not *posit a valid chain behind the certificate — a proved lemma reconstructs the prefix chain from the fold steps, so the chain's existence is a consequence, not a hypothesis. (Lean:* `GroundedCert`*.)*

**Assumption 5** (Clock)**.** *The verifier's clock is synchronized to within $\approx n/2$ slots. A clock running $\delta$ slots behind accepts chains up to $n+\delta$ slots stale; the forged-time bound keeps a fork harmless out to $\approx \frac{3}{2}n$*

*slots, so the single n-slot recency rule absorbs up to $\approx n/2$ slots of clock skew and network propagation combined — at the prototype's parameters, $\approx 14$ s for both together.*

**Assumption 6** (Honest delivery — liveness only)**.** *For the liveness theorem only: the honest producers of each newly maturing window were live, and their blocks reached the current producer in time to be built on. (Lean: `HonestBlocksCover`.)*

# 6 What the theorems guarantee

The results come in a fixed order. First, light-client safety (Theorem 1). It is proved about the protocol's structural core, and it covers the full validator because every chain the full validator accepts also passes those structural checks (`validChainK_structural`). Its counterpart at the certificate presentation, under the full validator with its key discipline and pin, is each mode's certified theorem in Section 6.3 (`keyrot_recent_certified_suffix_agreement` and its scheduled twin). Next, the timed model, where forged chains take real time (Theorem 2); this timed model is also where the recency scoping of Assumption 2 stops being an assumption and becomes a theorem (a separate derivation, `sigUnforgeableRecent_of_timed`). Then the rotation results per mode (Section 6.3), and finally liveness.

Two adversaries appear, and it matters which one a result faces. The *per-slot* adversary of Section 5 takes full control of any slots it likes within the budget, but walks away with no key it can reuse elsewhere. The *key-stealing* adversary of Section 6.3 additionally exports signing keys outright (defined there). Theorems 1 and 2 face the first; all of Section 6.3 faces the second.

Every result below is a Lean theorem, proved once against the model. The light-client and timed results (Theorems 1 and 2) are carried to each implementation by that implementation's soundness bridge (Section 7), which also covers the validator's key-discipline conjunct; the rotation-mode results of Section 6.3 and the liveness theorem are, today, proved at model level. A guarded `#print axioms` per headline theorem is checked in (`Molt/Axioms.lean`), so the build fails — rather than a reviewer noticing — if any proof ever assumes more than classical logic.

## 6.1 Light-client safety

**Theorem 1** (Recent chains share their deep blocks; Lean `light_client_safety`)**.** *Fix $n \geq 1$, a deployment genesis, and a verifier clock now. Suppose Assumptions 1–4 hold with the n-slot recency rule. Take any two certified chains accepted by the validator of Section 4, both with recent tips ($now \leq tip.slot + n$). Then any block of the first chain's suffix with at least n blocks above it equals the second chain's block at the same height, whenever the second chain's suffix exposes that height with n blocks above it too. In particular, all recent accepted chains agree on their common prefix up to n blocks below the lower tip, at every height both presentations expose.*

The "exposes that height" hypothesis is a restriction of presentation, not of operation: a certificate-plus-$n$-block chain can always be re-presented as a certificate$'$-plus-$2n$-block chain (grounding attests a claim at every fold), so a holder can re-expose as many blocks as a comparison needs.

Two remarks. First, the conclusion also covers a chain the node *produced* rather than received, because production ships nothing that does not pass the same validator (Section 4). Second, the proof *derives* honest-slot uniqueness instead of assuming honest behaviour: every block of an accepted certified chain either carries a checked signature (suffix) or is vouched for by the certificate's grounding (prefix), and Assumption 2 pins each honest-slot block to its producer's single signing call. From there it is the classic count: $2q > n + f$ in the trailing matured window forces the two chains through a shared honest block.

## 6.2 Forged chains take real time

The recency scoping in Assumption 2 needs a model in which coercion has a price. The *timed model* (`TimedExecution`) adds a real-time axis: a log of what each real slot's producer signed — at an honest real slot, at most its own current block; at a bad real slot, arbitrarily many blocks of the adversary's choosing under that producer's key — arbitrary content, and stamped with any slot at all, not only the real slot at which the signing happens. The id-formation contract of Section 3 appears as the field `chain_order`: a block can be signed only once its parent is *available*, because the parent's id preimage contains the parent's signature, and predicting another party's signature on a known message is exactly

an EUF-CMA forgery. (This holds even for a deterministic signature scheme: the value is determined, but computing it without the key is the forgery the game forbids.)

**Theorem 2** (Forged suffixes advance at half speed; Lean `forged_time_bound`)**.** *In the timed model with the fault budget: let a valid chain be available at real slot $r_{\text{now}}$, let $F$ be one of its blocks, first signed at real slot $r_0$, and let every chain block past $F$'s slot have been signed only at bad real slots (the suffix above $F$ is forged by coercion). Then*

$$q \cdot \left\lfloor \frac{tip.slot - F.slot}{n} \right\rfloor \;\le\; f \cdot \left( \left\lfloor \frac{r_{\text{now}} - r_0}{n} \right\rfloor + 1 \right).$$

*Since $q \approx 2f$: forging a suffix that spans $s$ slots takes about $2s$ real slots. The from-genesis corollary (`forged_chain_time_bound`) reads: a fully forged chain claiming tip slot $s$ cannot exist before real slot $\approx 2s$. (The Lean statements carry non-degeneracy side conditions — $n \ge 2$, $F$ not the first block; a forged span under one window makes the bound trivially true, ruling nothing out.)*

The proof is two counting facts. First, at most one block of a valid chain is first-signed at any real slot. At real slot $r$ only $r$'s producer's key can be made to sign, so two chain blocks first-signed at $r$ belong to one producer and their slot stamps differ by at least $n$; density puts some other seat's block between those stamps; and signing order (`chain_order`: a block cannot be signed before its parent exists) forces that middle block's first signing back to $r$ as well — under a key that real slot cannot reach. Second, density demands $q$ blocks per *stamped* window while the budget supplies only $f$ bad slots per *real* window. The ratio $q/f \approx 2$ is the theorem.

This is what makes the verifier's recency rule sound, with numbers: a fork meeting the $n$-slot recency bar carries at most $\approx 2f \approx 2n/3$ coerced blocks — short of the $n+1$ needed to fake an $n$-deep ancestor (the faked block plus the $n$ above it that make it trusted). The breakeven — the staleness at which half-speed forging has had time to mint $n+1$ blocks — sits near $\frac{3}{2}n$ slots, which is the $\approx 50\%$ margin quoted in Assumption 5. A patient adversary *can* grow a fork forever by harvesting coerced signatures window after window — but only at half speed, so its tip falls ever further behind real time, and a stale tip is exactly what the recency check refuses. This is also why Assumption 2(c) is derived rather than primitive: in the timed model it follows from plain EUF-CMA plus a *no-back-dating* residue — a signature bearing an honest slot's stamp was produced at that slot exactly, never stockpiled earlier and never minted later (`sigUnforgeableRecent_of_timed`; Appendix A gives the derivation, the residue's exact content, and its machine-checked independence).

## 6.3 Key rotation

The protocol's key-rotation machinery was defined with the protocol itself: the in-band version (`keyIndex`, Section 3) and the monotone rule (`keyMonoOk`) are already part of the one chain validity. What this subsection adds is the last conjunct a deployment fixes — the *pin*: a bound on the version a block may declare (a floor in modes 1–2, exact per grid window in mode 3), which is what makes a retired key *dead* rather than merely old. The three modes differ in exactly one thing — **what computes the pin**: the chain itself (mode 1), slot arithmetic (mode 2), or a roster-wide counter read off the signatures (mode 3). That one choice determines everything a light client must hold, so we state each mode as: the rule, what the operator must do, what the client must do, and what theorem you get.

**The adversary.** Everything in this subsection faces the key-stealing adversary: a stolen key of *any* version signs anything, at any slot, forever — no forward security [12] is assumed. Its *signature surface* — the bundle of custody and unforgeability assumptions the theorems consume, the analogue of Assumption 2 — is rescoped accordingly: the honest-slot uniqueness guarantee now holds only for non-stolen, non-rented entries. The budget (refining Assumption 1) is two rates per $n$-slot window. The first: $\rho$ *rented* slots — slots the adversary fully controls for that one slot and then loses, exactly Assumption 1's bad slots. The second: $T$ producers with a *then-live* key stolen — a stolen version at-or-above the floor where it is counted. Together: $\rho + T \le f$. The key-stealing corruption predicate (`badKeyrot`) counts a slot bad if it is rented *or* some version at-or-above the one in force for its producer is stolen — which builds *healing* into the definition: a stolen key stops counting the moment its retirement takes force. Each mode consumes its own rescoping of this one surface — the *key-stealing surface* (mode 1), the *scheduled surface* (mode 2), the *core surface* over the pin-free validator (mode 3) — and every rescoping includes *version independence*: stealing version $j$ yields nothing about $j' > j$, which key-insulated signatures [13] provide off the shelf. Hash injectivity (Assumption 3) is consumed alongside,

its domain rescoped per mode: any-registered-version signed blocks (`KeyStealingSigned`) for mode 1's chain-level rule, declared-version signed blocks (`SignedDeclared`) for modes 2–3 and the certified mode-1 form.

*Mode 0, formally: loss is not theft.* A *lost* key (destroyed, not exfiltrated) gives the adversary nothing; the loser rotates and the validator already supports it. With nothing stolen the corruption the theorems count collapses to rent alone in every mode (`badKeyrot_lossOnly`, `badSched_lossOnly`; mode 3 inherits mode 2's count) — no anchor, no cadence, any pin. This is the operational mode 0 of Section 2: whichever pin a deployment fixed, a genesis-and-clock client gets the core guarantees of Sections 6.1–6.2 unchanged.

**Mode 1: reactive rotation.** *The rule.* A producer rotates whenever it chooses, by signing under a higher version. The pin is read off the chain being validated: seat $i$'s version *in force* at slot $s$ (`inForce`) is the highest version $i$ has declared in the chain's *confirmed prefix* — blocks at least $n$ slots old — and no block may declare less (`validSignedChainK'`). Why $n$ slots: a rotation must take effect at one agreed moment on every chain, "the announcing block is confirmed" is that moment, and $n$ is exactly the depth at which all accepted chains provably agree on the confirmed prefix (the Lean proof takes this depth as a parameter and requires only that it be at least $n$). At the prototype's parameters a rotation takes force in under half a minute. Taking force and the client's sync cadence below are separate clocks.

*Why an anchor is needed at all.* On any chain that contains a rotation, the retired key is dead by the pin. The residual threat lives elsewhere: a fork built from *before* the rotation, on which the floor never rose and the old key is still current — so a key stolen even years later is raw material for a self-consistent *fake past*. No validator rule can reject such a fork from the inside; it can only be excluded by the client knowing one thing the fork cannot contain: a sufficiently recent block of the real chain. That is the *anchor* — a block from the client's trusted region, i.e. at least $n$ deep below its last verified tip, or a checkpoint — with chains not containing it refused. (Anchoring the literal tip would be a mistake: only $n$-deep blocks are guaranteed shared, and a tip the network never adopts would wedge the client against the real chain.) This is a weak-subjectivity assumption [5], but a narrow one, and Theorem 3 is its exact price tag.

**Theorem 3** (The client sync rule; Lean `sync_rule`, `sync_rule_mem`)**.** *Let a light client re-verify a chain* at least once every $n$ slots*, each time taking as its anchor the block $n$ deep below the newly verified tip. Suppose the key-stealing surface and hash injectivity hold, and the $\rho/T$ budget — read off the lower accepted chain's confirmed floors — holds on every $n$-slot window starting in the trailing $5n$ slots, or later. Then any two accepted, recent chains longer than $n$ blocks that both contain the anchor agree: at equal tip heights, on the block $n$ below each tip; at unequal heights, the lower chain's $n$-deep block is a block of the taller chain, at least $n$ deep there.*

In one sentence: **sync at least every $n$ slots — the same $n$ as the recency rule — and no fabricated past can ever be accepted, at the price of one standing assumption: over the last $5n$ slots — about two minutes at the prototype's parameters — rented slots plus producers suffering a theft of a then-live key total at most $f$ per window.**

The $5n$ is the reach of that assumption: how far back the anchor can sit, plus the one window the counting needs. Each summand is one fact. At sync time the freshly verified tip passed the recency rule, so it is at most $n$ slots old. The anchor sits fewer than $2n$ *slots* below that tip although it is $n$ *blocks* deep — density leaves no room for more (two full windows in the gap would demand $2q > n$ blocks where only $n$ exist; Lean `deep_block_span`). And the client returns within $n$ slots of the sync. Total anchor age: under $4n$; plus the one window the budget's counting needs: $5n$. (The "or later" in the budget clause matters only against chains dated beyond the verifier's clock, which a real clock rejects; nothing at all is assumed about windows starting earlier.)

Two readings of the budget clause are worth fixing in place. First, it asks for strictly less than the natural lifetime assumption: a deployment that keeps $\rho+T$ within $f$ on every window from genesis onward satisfies the hypothesis outright, so Theorem 3 covers it as an immediate special case — and taking the anchor to be the genesis block recovers the lifetime-budget statement exactly (`client_refresh_rule` at $A :=$ genesis). The trailing scope exists because, under key theft, the lifetime form is the one a deployment cannot defend — the next paragraph makes this precise: the census (the per-window count of stolen producers) is retroactive, so a lifetime budget quietly demands that retired keys stay secret forever, whereas the trailing form confines the demand to two minutes. Second, the trailing $5n$ slots are not a bootstrap condition: the window slides with the verifier's clock, re-anchored at every sync — it

measures the client's own recency of state, not the chain's age, and an old chain is exactly as protected as a young one.

Why the cadence cannot simply be enlarged: *the budget's census is retroactive.* The model carries no theft times, so the census at a window counts every producer whose version in force there is ever stolen and not yet retired by then — a theft is charged to the windows *before* it as well, for as long as the same version was in force, and rotating afterwards never subtracts from them. The oldest windows of a stretch therefore carry the *total* theft count of the whole stretch, and the longer the stretch, the larger that total. A client syncing every $10n$ slots would therefore be assuming "at most $T$ producers stolen across $\approx 14n$ slots *in total*" — and an adversary that quietly collects one key every couple of windows, each victim rotating promptly so that no more than one is ever compromised *at a time*, breaks that budget — and with it, every guarantee this development can state. Syncing every $n$ costs a $5n$ stretch — one window above the design's hard floor of $\approx 4n$, which only continuous re-verification approaches, since the anchor alone is up to $\approx 3n$ old even for an always-online client. The standing assumption is then: *at most $T$ producers suffer a theft of a then-live key per $\approx 5n$ slots — a total per stretch, not a concurrency bound.* The operator's half of the rule reads the same way: keep rented slots plus thefts within $f$ per window over the trailing $5n$ slots, remembering that a theft sits on the books of every earlier window of its stretch. The agreement itself also holds at the certificate presentation, with the floor — the map sending each seat to its version in force — carried as a snapshot in the claim — the certificate must attest claim and floor *together*, an unauthenticated floor readmits stolen retired keys (`keyrot_recent_certified_suffix_agreement`, over the strengthened grounding `GroundedCertK`) — but under the global, all-window budget: carrying the anchored, trailing-$5n$ form to certificates is future work.

*The maximum sync period, both directions formal.* Write $F$ for the client's sync period. *Upward* (`max_sync_period`): safety holds whenever rented slots plus the theft census stay within $f$ on every window starting in the trailing $F + 4n$ slots or later. So a deployment that can honestly cap $\rho$ plus ever-stolen, not-yet-retired-there producers at $f$ per window over every such stretch of $S$ slots supports

$$F_{\max} \;=\; S - 4n.$$

*Downward* (`census_accumulates`, `census_accumulates_later_thefts`, `no_budget_beyond`): the census at a window is *at least* the number of distinct producers with an ever-stolen version not yet retired there — a theft at or after a window charges that window, because floors never fall (`inForce_mono`), and healing never subtracts — so one victim past $f$ in a stretch leaves no budget any deployment could satisfy: beyond $F_{\max}$ the safety theorem's precondition is not merely weaker — nothing can meet it, so the theorem promises nothing. "Exactly" should be read at that level: past $F_{\max}$, slot for slot, the precondition can no longer be met. The smallest $F$ at which a full forged execution is *known* to exist lies somewhat higher — by $\approx n$ slots of span, and in victims. Density binds every fork window, so if honest producers only ever extend the real chain, an accepted fork needs $\approx q \approx 2f$ victims whose retirements postdate the divergence (pre-divergence rotations are inherited through the anchor and stay dead); in the bare model, where nothing constrains which branch honest producers extend, the count falls toward $f + 1$. Certifying one such execution end-to-end in Lean is future work. A deployment prepared to assume strictly more — honest producers never extending adversarial branches, and theft *times* inside the signature assumptions, so that a stolen key is charged from the moment it is stolen rather than to every window its version was in force — could in principle support a larger $F$; that stronger model is outside the present development (future work; honest scope below). The sync rule is the $F = n$, $S = 5n$ instance. (Engine: `client_refresh_rule`, parametric in the anchor's age; `stay_recent_client_safe` derives the age bound.)

One consequence should be said plainly: mode 1's client is an *active* client — at the prototype's parameters, a sync every $\approx 28\,\mathrm{s}$ — which is the natural fit for services, bridges, and validators acting on the chain continuously. A client that sleeps for hours or weeks does not tune a constant; it runs mode 2 or 3, which need no anchor at all.

Three consequences fix interpretations that are easy to get wrong.

- *Rotating away from a possibly-stolen key assumes nothing about that key's secrecy*: an exposed then-live key is one of the $T$ thefts the budget already tolerates — healed for later windows once its rotation confirms, still on the books of the stretch's earlier windows.

- *Keys retired more than $5n$ slots ago need no secrecy at all*: the budget starts at the trailing $5n$, so such a key can leak or be published without harming any sync-rule client; within the stretch, retired-key secrecy is not a separate duty but part of the same $T$ budget.

- *Rotation frequency and client duties are decoupled*: no client re-anchors, re-syncs, or updates in response to a rotation; the sync cadence is scoped by time, never by events.

**A realistic mode-1 deployment, end to end.** Seven seats ($n = 7$, $\tau \approx 4\,\text{s}$; $f = 2$, $q = 5$), each signing behind a hardware module with theft alarms; on suspicion a seat signs its next block under a higher version, in force $n$ slots after that block — at most about a minute from the suspicion, counting the wait for the seat's own slot. Clients are services acting on the chain continuously: each joins from a deployment-published checkpoint (its first anchor), then re-verifies at least every 28 s, anchoring the $n$-deep block of each verified chain. What must then be true of the world: the key-stealing surface and hash injectivity (the adversary paragraph's rescopings of Assumptions 2–3); the standing clock assumption (Assumption 5); and *over any* $\approx 2$*-minute stretch, rented slots stay within* $\rho$ *per window and at most* $T$ *of the seven seats suffer a theft of a then-live key,* $\rho + T \leq 2$ — a catastrophic-compromise ceiling, not an operating point, and one a monitored deployment can attest because the stretch is two minutes, not a history. Conclusion, machine-checked (`sync_rule`, `sync_rule_mem`): any two chains such a client accepts at a sync — both, by its rule, containing its anchor — agree: at equal heights on the block $n$ below each tip, and the lower tip's $n$-deep block lies on every taller one — hence, block by block through the parent-id chain (`same_block_same_prefix`), on the lower chain's whole confirmed prefix. Agreement with the honest chain itself follows by an induction over syncs argued on paper, not itself machine-checked. Each sync's agreement keeps the next anchor on the honest chain while the honest tip is at least as tall as the accepted one (density keeps a recent taller fork's $n$-deep block below its divergence, hence on the honest chain — an informal argument). The join checkpoint starts the induction as the op-model's fresh trust event: a checkpoint no older than $\approx 4n$ slots satisfies the general Lean form `client_refresh_rule`. And the honest tip staying recent is a liveness-style side condition, outside the safety hypotheses. At the full-chain presentation this is the complete story.

**Mode 2: scheduled rotation.** *The rule.* The version in force at slot $s$ — modes 2 and 3 call it the *generation* — is a protocol constant $gen(s) = \lfloor s/R \rfloor$: a pure function of the slot, identical on every chain, pinned at the verifier and never peer-supplied. The validator adds one check: $gen(slot) \leq keyIndex$ (`validSignedChainSched`). Because the pin reads nothing off any chain, a fork *cannot disagree* with the real chain about which keys are retired — the fake-past construction above is structurally impossible, and the anchor has nothing left to do.

**Theorem 4** (Scheduled safety; Lean `scheduled_client_safety`)**.** *Under the scheduled validator, the scheduled signature surface, hash injectivity (Assumption 3, over declared-version signatures), and a* chain-independent *$\rho/T$ budget: two accepted chains rooted in one genesis with recent tips agree: at equal heights, on the block $n$ below each tip; at unequal heights, the lower chain's $n$-deep block is a block of the taller, at least $n$ deep there (`sched_recent_tip_ancestor_mem`). No anchor, no sync rule, no confirmation wait: the client holds genesis, a clock, and the constant gen, and may stay offline arbitrarily long.*

*Operator duties.* Keep one offline *cold root* per seat and delegate each generation's signing key just in time for its era; do not expose a generation's key before its era approaches. Two of these duties are *named operational assumptions the cited theorems do not consume*: *not-before* — a generation's key cannot be stolen before it is derived; formalized separately as `NoPrematureTheft` in the timed theft layer — and *cold-root custody*, which stays informal. They matter because the census counts *any* stolen not-yet-retired version, future generations included, against every window until that generation's era ends: a key derived early, or a stolen cold root, consumes the budget for every era it covers, and the $T/R$ sizing holds only under the just-in-time discipline. The era length $R$ is a deployment dial: longer eras mean rarer cold-root ceremonies but a longer exposure window for a stolen current key. No erasure is required — a retired key is harmless *by rule*, not by secrecy: a chain whose tip declares a generation whose era ended more than $n$ slots ago fails the recency check (`sched_oldkey_fork_stale`). The statements carry to the certificate presentation with no floor snapshot in the claim — the certificate state stays the plain one, each fold merely adding the pin check — since the pin is computed from each block's own slot (the grounding `GroundedCertSched`; `sched_recent_certified_suffix_agreement`). *What breaks first:* (i) a peer-supplied schedule — the pin must be a verifier constant, or a zero schedule from the network readmits every retired key; (ii) theft of the *current* generation, which signs freely until $n$ slots past its era's end — size $T$ and the era length $R$ together; (iii) a stolen *cold root* — every future era at once.

**A realistic mode-2 deployment, end to end.** At the ceremony: each seat generates its cold root offline and the deployment publishes $gen(s) = \lfloor s/R \rfloor$ with, say, $R =$ one week of slots. Weekly, each seat's cold root signs one delegation for the coming generation and goes back in the safe; the live signing key never predates its era by more than the provisioning margin. Clients are wallets holding genesis, a clock, and *gen*; they sleep for months and sync on wake. What must then be true of the world: the scheduled surface and hash injectivity (the adversary paragraph's rescopings); the standing clock assumption (Assumption 5); not-before and cold-root custody (the ceremony discipline above); and *at most $T$ of the currently live generation keys stolen per window over the trailing $\approx 2n$ slots, $\rho + T \leq f$* — the horizon-scoped budget, with no retroactive reading: no slot more than $2n - 1$ before the clock is ever budgeted, and genesis agreement becomes a consequence rather than a hypothesis (`sched_recent_tip_ancestor_agreement_horizon`). The week-long exposure window per stolen live key is what $T$ is sized against. Conclusion, machine-checked: any two accepted chains agree at equal heights on the block $n$ below each tip — hence, block by block through the parent-id chain (`same_block_same_prefix`), on the whole shared prefix — whatever happened while the wallet slept; at unequal heights the same holds horizon-scoped (`sched_recent_tip_ancestor_mem_horizon`). Only the certificate form is proved under the global all-window budget, with the strengthened grounding `GroundedCertSched` (`sched_recent_certified_suffix_agreement`); carrying the horizon scoping to certificates is future work.

**Mode 3: free-cadence lockstep.** *The rule.* No schedule anywhere. The roster advances a single shared generation counter at moments of its own choosing — an incident, a governance decision — and validity enforces only the *discipline* of the switch (`noMixing`): within each grid window (slots $[Wn, Wn{+}n)$; the fixed grid, not a sliding window) all blocks declare one generation, and the counter never decreases across grid windows (`validSignedChainLock`). The verifier never learns the cadence; it reads each grid window's generation off the signatures themselves.

**Theorem 5** (Lockstep safety; Lean `lockstep_client_safety`)**.** *Under the lockstep validator and the lockstep hypothesis package (the roster tracks one monotone counter; honest signatures declare their grid window's value; the genesis declares the counter's initial value; hash injectivity; the same $\rho/T$ budget, assessed at the one-grid-window-lagged counter over the pin-free core validator's signature surface): the same anchor-free agreement as Theorem 4, in both forms (`lockstep_recent_tip_ancestor_mem` at unequal heights). The client holds even less than mode 2's — genesis and a clock, with no schedule to store at all.*

The route is a machine-checked *pinning* theorem (`lockstep_declares_rosterGen`): on any accepted recent chain, every matured grid window declares exactly the roster's counter — a grid window at any other generation would need a quorum of adversarial blocks, which a $\rho + T \leq f$ adversary cannot supply. Pinning makes every accepted lockstep chain pass mode 2's validator at the (one-grid-window-lagged) counter, so the whole scheduled theorem set transports.

*Operator duties.* Two, and they are the mode's real price: *lockstep coordination* — all honest signers track the one counter (a split here is an operational failure the theorems do not cover) — and *real erasure* of retired key material at each switch (its budget credit: honest scope below). Erasure that is not real is silent to the protocol: keys held in an HSM or only ever in RAM are defensible; flash storage, which does not erase by overwriting, is not.

**A realistic mode-3 deployment, end to end.** Each seat signs from a hardware module whose retired generations are destroyed on switch; the genesis block declares the counter's initial value; switches are agreed out of band — an incident response, a governance action — and every honest signer moves at the agreed grid-window boundary. Clients hold genesis and a clock, nothing else, and sleep arbitrarily long — verifying the full signed chain on wake, since the certificate presentation is future work. What must then be true of the world: the core surface over the pin-free validator and hash injectivity (the adversary paragraph's rescopings); the standing clock assumption (Assumption 5); the coordination discipline above (a silent split among honest signers is an operational failure outside the theorems); and *the same $\rho/T$ budget as mode 2's global form, assessed at the one-grid-window-lagged counter.* Conclusion, machine-checked (`lockstep_client_safety` and `lockstep_recent_tip_ancestor_mem`, via the pinning theorem `lockstep_declares_rosterGen`): any two accepted chains agree — at equal heights on the block $n$ below each tip, at unequal heights the lower chain's $n$-deep block lying on the taller — hence, through the parent-id chain (`same_block_same_prefix`), on the lower chain's whole confirmed prefix — with no schedule anywhere and no anchor.

**The modes at a glance.**

| | pin computed from | client holds | erasure | anchor |
|---|---|---|---|---|
| 0 no theft | any of the three | genesis + clock | no | no |
| 1 reactive | chain floor, $n$ slots deep | genesis, clock, anchor, $n$-slot sync | no | yes (Thm. 3) |
| 2 scheduled | slot arithmetic $gen(s)$ | genesis + clock + $gen$ | no | no |
| 3 lockstep | roster counter, no-mixing | genesis + clock | yes | no |

Read the trust downward: mode 1 trusts the client's own recency of state; mode 2 trusts slot arithmetic plus key-provisioning discipline; mode 3 trusts roster coordination plus physical destruction. Modes 1–3 share the $\rho + T \leq f$ budget and a recency-scoped signature surface; mode 0 is any of them at $T = 0$.

**Honest scope.** Three bounds on what this subsection claims. First, the budget counts a key as stolen in every window where it was in force, *even if the theft only happens later* (within the trailing $5n$ slots, for mode 1); distinguishing thefts before a block's creation from thefts after it would need creation times inside the signature assumptions, and is future work. Second, mode 3's budget still counts thefts across all generations together, exactly as mode 2's does — letting erasure cap each generation's count separately is designed but not yet formalized. Third, mode 3's theorems are about full chains; their certificate-level form is future work.

### 6.4 Liveness, and the slot time

Safety bounds what a verifier can be made to accept; liveness is about progress, and its proofs use only the structural core of the paper's single chain validity (Section 3) — no rotation hypothesis enters.

**Theorem 6** (Liveness; Lean `production_liveness`, `global_liveness`)**.** Local: *under the fault budget and honest delivery (Assumptions 1 and 6), when a slot arrives that belongs to an honest producer holding an accepted chain, production succeeds — the extension passes the same validator everyone runs.* Global: *a synchronous honest run within the budget (over the run's span; each honest block reaches the next honest producer within its slot) is validator-accepted with one block per honest slot, so the chain reaches any height and every block eventually becomes $n$-deep.*

The fork-choice rule — adopt the valid chain with the highest tip — is what makes honest blocks coalesce into the single chain the global form assumes; a fully multi-chain network model is future work.

One quantity remains: the slot duration $\tau$. It is set by prover economics, not network latency: the slot must be long enough for the proving pipeline to keep its uncovered suffix under $n$ blocks forever. The Lean development proves a threshold that is sufficient, and necessary over the pipeline's steady states (`slot_time_sufficient`, `slot_time_necessary`); we do not reproduce the formula here — Section 7 instantiates it at the measured proving costs.

## 7 Implementation and measurements

**One source, three instantiations.** The reference implementation is one Rust file, `rust/src/lib.rs`, written in the fragment that the Charon + Aeneas toolchain translates to Lean 4. From it:

- *The native validator* is imported into Lean and proved sound against the model: if the extracted code accepts a chain, the semantic predicate holds (`rust_valid_chain_sound` and companions). Soundness is the only direction safety needs — completeness is a liveness concern — and it is what carries Theorems 1 and 2 to the shipped code (`rust_recent_tip_ancestor_mem`). The bridges cover the validator's structural core and its key-discipline conjunct (`rust_valid_chain_sound`, `rust_valid_chain_k_sound`); the three rotation *pins* of Section 6.3 are, today, stated and checked at model level.
- *The circuit* instantiates the same source's backend-generic validator (`valid_chain_be`) over plonky2, so the recursive certificate proves exactly the predicate that was verified (`valid_chain_be_validChain`); the per-gadget faithfulness of the circuit backend is the one remaining trust assumption, stated rather than proved.

| operation | time | circuit rows |
|---|---|---|
| consensus validator alone (8-block chain) | 0.035 s | $2^9$ |
| one in-circuit signature verification (subproof) | 0.36 s | $2^{13}$ |
| verifying one subproof (recursion floor) | 0.19 s | $2^{12}$ |
| certificate, 8 signatures inline | 6.6 s | $2^{16}$ |
| certificate, 8 signatures as recursive subproofs | 3.0 s | $2^{15}$ |
| 2048 subproofs, balanced tree (critical path) | 5.0 s | $2^{13}$/node |

Table 2: What the prototype measures, and why it means the design is viable: the verified consensus rule is <2% of any signature-bearing certificate (row 1 vs. rows 4–6); post-quantum signatures dominate and recurse well — a signer's subproof is verified at the recursion floor, and a balanced tree keeps latency logarithmic in approvals per slot (5.0 s at 2048, against ≈770 s folding sequentially) while hardware grows linearly.

- *A TypeScript twin* (`moltPetit.ts`) reaches Lean through an independent compiler (Thales) with its own soundness bridge; the vendored emission hand-corrects three recorded emitter bugs (`tools/thales-reemission/`), a hand-audited step in this pipeline's trusted base. It shows the methodology is not tied to one language, and keeping two implementations honest against one model is what kept the model abstract.

The trusted base is exactly: the Lean kernel; the pinned translators (Charon + Aeneas, Thales together with the recorded hand-corrections to its emission, the circuit gadgets); and the faithfulness of the named assumptions to the deployed primitives. Crypto enters only as theorem hypotheses, and every handle (keys, signatures, certificates, payload commitments) is opaque to the protocol — which is also why no numeric-model axiom of either host language reaches any theorem.

**Measurements.** The prototype bundles the consensus rule with a minimal application — a committed key set evolving by signed rotations — on plonky2 [25] over its native 64-bit field (Goldilocks) with Poseidon hashing [17], signing with SPHINCS+ [14, 15] over Poseidon (parameter set `128s`). All numbers are wall-clock on one Intel i5-13500 (14 cores, 20 threads). "Circuit rows" is the standard size measure of a statement handed to the proof backend — proving cost grows with rows — and the *recursion floor* is the fixed minimum cost of verifying any subproof inside another proof.

Two readings matter for deployment. First, *consensus is free*: the machine-checked validator adds no appreciable proving cost on top of whatever the application already pays for signatures. Second, the measured proving costs feed the slot-time rule of Section 6.4, which at this hardware and $n = 7$ recommends $\tau \approx 4$ s — the figure used throughout the paper. Where a deployment tolerates stateful signing, XMSS [16] over the same Poseidon gadgets shrinks the in-circuit verifier from $2^{13}$ to $2^{10}$ rows and speeds full certificates by roughly 10×, at the cost of one-time keys whose bookkeeping the consensus layer already maintains.

# 8 Related work

**Density in proof-of-stake.** The window-density rule's nearest relatives are in the Ouroboros family, where density is a first-class quantity: Genesis [4] resolves deep forks by comparing growth just after the last common block (a bootstrap that replays history), and Samasika [3] — Mina's consensus — carries a constant-size minimum-window-density summary, the closest prior pairing of density with succinctness. The structural difference: in every Ouroboros variant density drives *fork choice*, so finality is probabilistic and depth-dependent, while Molt Petit makes density a hard *validity predicate*, giving deterministic finality at a fixed depth of $n$ blocks. Molt Petit also fixes an equally weighted roster with the public round-robin leader $s$ mod $n$, where the Ouroboros line is stake-weighted, with private VRF-based election from Praos [2, 1] onward — and its security argument is a machine-checked artifact, not a hand-written analysis.

**Succinct certificates and light clients.** The constant-size recursive certificate is anticipated by Mina [6], whose recursively composed SNARK attests the *whole* history; Molt Petit's certificate is instead actionable only under the recency rule — recency-scoped rather than whole-history agreement. Plumo [7] syncs ultralight clients by SNARK-proved committee hand-offs, with a pen-and-paper argument

over a committee-weighted set. The proof-of-work line is probabilistic by nature: NiPoPoWs [8] via rare superblocks, FlyClient [9] via random sampling; both certify accumulated *weight*, where Molt Petit's certificate is a single recursive proof of a deterministic validity predicate — constant-size and sampling-free.

**Deterministic-finality BFT.** Molt Petit shares with Tendermint [10] and HotStuff [11] a fixed participant set, rotating leaders, and deterministic finality — by the opposite mechanism. They finalize through explicit quorum-certificate *voting*; Molt Petit exchanges no votes at all: the same $2n/3$ threshold is enforced by counting blocks in a window, and the light client's constant-size object is a recursive proof of that counting rule, not an aggregate of votes.

**Machine-checked consensus.** Velisarios [18] (BFT in Coq), Verdi [19] (verified Raft, extracted OCaml), and IronFleet [20] (Paxos-based replication in Dafny) all prove running consensus code correct. Molt Petit differs in the object — a vote-free density consensus carrying a recursive light-client certificate — and in the pipeline's *direction*: those systems refine or extract downward from a prover-internal model, while Molt Petit *imports* the deployed sources into Lean [21, 22] via Charon + Aeneas [24] and Thales [23], so the theorems are about the already-runnable artifact. The trusted translation step does not disappear — the front-ends stand exactly where extraction stands in theirs — but it points from the shipped code into the prover rather than out of it.

# 9 Limitations and future work

**What the design gives up.** Four properties of the surveyed systems are deliberately traded away, and a deployment should know it is buying their absence:

- *Dynamic participation.* The roster is fixed and closed: no reconfiguration, no membership churn. Signing keys rotate (Section 6.3); seats do not.
- *Stake.* The set is equally weighted, the leader schedule public and deterministic — accepting the censorship exposure of a predictable next leader in exchange for needing no randomness beacon.
- *Liveness recovery.* There is no view change and no fork-choice repair. If a matured window ever falls below $q$ blocks, the deficit can never be back-filled and the chain halts permanently, by design; the liveness theorems are conditional on the budget that has then failed, while safety is untouched. Ground this: at $n = 7$, $\tau \approx 4\,\text{s}$, three seats missing their slots within one $\approx 28\,\text{s}$ window halts the deployment forever.
- *Asynchronous safety.* Safety itself is clock-scoped: the recency rule is a clock check, so Assumption 5 is load-bearing for safety, not only liveness. What is never assumed is network synchrony — delivery is fully adversarial in every safety theorem.

The trade is plain: a closed, equally weighted set with no recovery is what lets finality be a validity predicate decided at depth $n$, rather than a fork-choice metric resolved probabilistically over a longer timescale.

**Named seams.** Beyond the trusted base of Section 7 — the Lean kernel; the pinned translation front-ends (Charon + Aeneas, Thales), including the hand-audited emitter-bug corrections in the vendored TS emission; and the per-gadget faithfulness of the circuit backend, stated rather than proved (the equivalence lemma `valid_chain_be_validChain` is proved at the native u64 instantiation) — everything not machine-checked is a named hypothesis standing next to the theorem that consumes it: the operational assumptions of Section 5; the key-stealing signature surface (assumed, not derived from the timed model); version independence of keys under theft (instantiable with key-insulated signatures [13]); the circuit path's out-of-circuit trust anchor (the roster commitment to the cold roots); and the recency check itself, which lives in the unverified node loop since the pure validators carry no clock — threading *now* through them is mechanical future work; and the per-mode operational assumptions of Section 6.3 (not-before and cold-root custody; lockstep coordination and erasure).

**The long-range residual, and options against it.** A patient adversary can bank coerced signatures into an arbitrarily long fork — forever stale, hence harmless to any client obeying the sync rule of Theorem 3, but never impossible. Three options would remove the residual rather than contain it, each at a price: forward-secure signatures [12] (a stateful signer); monotone, once-corrupt-always-corrupt accounting (simply a weaker adversary); or running rotation modes 2–3, which remove the anchor hypothesis outright at the price of a schedule or an erasure discipline. Consensus-managed one-time keys (the XMSS variant of Section 7) shift key state into the chain and are complementary. We keep the per-slot model as the most adversarial and operationally simplest, and accept the contained residual.

**Artifact.** The repository builds with `lake build`: the model and results, both implementation bridges, and the paper-facing `Molt` library with its axiom guards. A fresh checkout re-verifies every theorem cited in this paper with that one command. That the vendored translations are what the pinned front-ends (Charon + Aeneas; Thales) emit from `rust/src/lib.rs` and `moltPetit.ts` is re-checked separately by the repository's `nix flake check`, which asserts faithfulness only and says nothing about the proofs. What no machine re-checks is exactly the trusted base above: the translators and circuit gadgets themselves, and the hand-audited re-emission corrections.

# A The honest-slot uniqueness assumption, derived

Assumption 2(c) is the one hypothesis in this paper that can look arbitrary on first meeting: *at an honest slot, a verifying block is the unique block its producer's honest signing path signed for that slot.* This appendix shows it is not arbitrary — in the timed model it is a theorem, and the model that yields it is the intuitive one: a block's *stamp* (the slot it claims) and the *real time* it was signed are separate quantities, and everything turns on how far apart they can be.

The timed model (Section 6.2) records what each real slot's producer signed: at an honest real slot, at most its own current block; at a bad one, anything under that producer's key. Assumption (c) instead speaks of a *stamped* log — per producer, per claimed slot. A projection connects them (`projectSigned`): a producer's entry for stamped slot $s$ is the block *stamped* $s$ that it signed *at real slot* $s$, its own slot — single-valued at an honest slot because an honest node signs at most once per slot (at a bad slot the entry is an arbitrary choice the theorem never consults). Two hypotheses then close the gap:

- the *EUF-CMA bridge*: a verifying signature was produced at *some* real slot — with no constraint on which;
- *no back-dating* (`NoBackdate`): a block whose stamp is an honest slot was signed *only* at the real slot equal to its stamp.

**Theorem 7** (Uniqueness, derived; Lean `sigUnforgeableRecent_of_timed`)**.** *In any timed execution satisfying the EUF-CMA bridge and no back-dating, Assumption 2(c) holds for the projected log — at every recency window, the bar being a parameter the proof never uses.*

The proof is three steps: the verifying block was signed at some real slot; no-back-dating forces that slot to be its stamp; so it sits in the log at its own honest slot, where signing-at-most-once makes it unique. Notably, recency is not even consumed — the per-block pinning comes entirely from no-back-dating, while recency separately buys the forged-time bound of Theorem 2.

*What no-back-dating is, and why it is not free.* Operationally it is key custody re-anchored to the stamp: an honest producer's slot-$s$ signature can only be its genuine slot-$s$ call, *and no bad slot of the same producer may mint a block bearing that honest stamp.* The constraint is on the *stamp*, not on the signer: a rented or stolen key still signs arbitrary content at any real slot, stamped with any slot it likes — any slot except one of its own producer's honest ones. The second clause is the whole content — the bare timed model does allow a bad real slot to mint a block stamped for one of its producer's honest slots, and we machine-check exactly such an execution (`noBackdate_independent`, at $n = 2$): every timed-model field holds, no-back-dating fails. So neither recency, nor the id-formation order, nor the forged-time bound supplies it; it is the precise extra hypothesis, checked to be *independent* — no timed-model field implies it. In the standard vocabulary this stamp-to-slot binding is two-sided: forward security proper [12] supplies the back-dating half (an evolved key cannot sign earlier periods), while the forward half — no pre-signing of future honest stamps — rests on the signing path's period being locked to real slots; a key-evolving scheme under a one-period-per-slot schedule, honestly updated, provides both halves — provided a bad slot yields signatures but never the key itself, since an exfiltrated evolving key can be evolved forward and defeats the forward half.

# References


[1] A. Kiayias, A. Russell, B. David, R. Oliynykov. *Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol*. CRYPTO 2017.

[2] B. David, P. Gazi, A. Kiayias, A. Russell. *Ouroboros Praos: An Adaptively-Secure, Semi-synchronous Proof-of-Stake Blockchain*. EUROCRYPT 2018.

[3] J. Bonneau, I. Meckler, V. Rao, E. Shapiro. *Ouroboros Samasika: A Proof-of-Stake Consensus Protocol for Succinct Blockchains*. O(1) Labs / Mina Protocol technical paper, 2020.

[4] C. Badertscher, P. Gazi, A. Kiayias, A. Russell, V. Zikas. *Ouroboros Genesis: Composable Proof-of-Stake Blockchains with Dynamic Availability*. ACM CCS 2018.

[5] V. Buterin. *Proof of Stake: How I Learned to Love Weak Subjectivity*. Ethereum blog, 2014.

[6] J. Bonneau, I. Meckler, V. Rao, E. Shapiro. *Mina: Decentralized Cryptocurrency at Scale*. Whitepaper, 2020. IACR ePrint 2020/352.

[7] P. Vesely, K. Gürkan, M. Straka, A. Gabizon, P. Jovanovic, G. Konstantopoulos, A. Oines, M. Olszewski, E. Tromer. *Plumo: An Ultralight Blockchain Client*. Financial Cryptography 2022.

[8] A. Kiayias, A. Miller, D. Zindros. *Non-Interactive Proofs of Proof-of-Work*. Financial Cryptography 2020.

[9] B. Bünz, L. Kiffer, L. Luu, M. Zamani. *FlyClient: Super-Light Clients for Cryptocurrencies*. IEEE S&P 2020.

[10] E. Buchman, J. Kwon, Z. Milosevic. *The latest gossip on BFT consensus*. arXiv:1807.04938, 2018.

[11] M. Yin, D. Malkhi, M. K. Reiter, G. Golan Gueta, I. Abraham. *HotStuff: BFT Consensus in the Lens of Blockchain*. PODC 2019.

[12] M. Bellare, S. K. Miner. *A Forward-Secure Digital Signature Scheme*. CRYPTO 1999.

[13] Y. Dodis, J. Katz, S. Xu, M. Yung. *Key-Insulated Public Key Cryptosystems*. EUROCRYPT 2002.

[14] D. J. Bernstein, A. Hülsing, S. Kölbl, R. Niederhagen, J. Rijneveld, P. Schwabe. *The SPHINCS+ Signature Framework*. ACM CCS 2019.

[15] National Institute of Standards and Technology. *Stateless Hash-Based Digital Signature Standard*. FIPS 205, 2024.

[16] A. Hülsing, D. Butin, S. Gazdag, J. Rijneveld, A. Mohaisen. *XMSS: eXtended Merkle Signature Scheme*. RFC 8391, 2018.

[17] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, M. Schofnegger. *Poseidon: A New Hash Function for Zero-Knowledge Proof Systems*. USENIX Security 2021.

[18] V. Rahli, I. Vukotić, M. Völp, P. Esteves-Verissimo. *Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq*. ESOP 2018.

[19] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, T. Anderson. *Verdi: A Framework for Implementing and Formally Verifying Distributed Systems*. PLDI 2015.

[20] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, B. Zill. *IronFleet: Proving Practical Distributed Systems Correct*. SOSP 2015.

[21] L. de Moura, S. Ullrich. *The Lean 4 Theorem Prover and Programming Language*. CADE 2021.

[22] The mathlib Community. *The Lean Mathematical Library*. CPP 2020.

[23] J. Alama. *Thales: a TypeScript-to-Lean compiler*. `https://github.com/jessealama/thales`.

[24] S. Ho, J. Protzenko. *Aeneas: Rust Verification by Functional Translation*. ICFP 2022. (With the Charon frontend, `https://github.com/AeneasVerif`.)

[25] Polygon Zero. *Plonky2: Fast Recursive Arguments with PLONK and FRI*. `https://github.com/0xPolygonZero/plonky2`.