LP-221: STARK-FRI Verifier Precompile

A dedicated EVM slot for strict-PQ STARK proofs, orthogonal to the P3Q signature dispatch at 0x012205

Zach Kelling

Lux Industries

2026-06-04

Abstract

LP-221 specifies the dedicated EVM precompile at slot `0x012220` for verifying strict-PQ STARK proofs. The verifier is a Plonky3 fork with the classical-curve surface stripped out: **FRI** [1] low-degree testing over the **Goldilocks** 64-bit prime field, **cSHAKE256** (FIPS 202 / SP 800-185) [14, 15] Merkle commitments, no KZG, no pairings, no elliptic-curve arithmetic. The reference implementation at `lux/precompile/starkfri/` (5 files, 22 tests passing under `-race` in 1.376 s on darwin/arm64) registers under config key `starkfriVerify`. LP-221 is the dedicated allocation that LP-300 §"Post-quantum precompile registry" had recorded as the displaced-Plonky3-fork placeholder pending dedicated LP allocation. LP-221 is orthogonal to LP-218: LP-218 owns slot `0x012205` for PQ *signature* verification (Pulsar / Corona / Magnetar kind-byte dispatched); LP-221 owns slot `0x012220` for PQ SNARK *proof* verification. One slot for signatures, one slot for SNARK proofs, two disjoint use cases, zero overlap. The on-chain wire format pins `VersionV1=0x01`, `MagicHeader="P3Q1"` preserved verbatim from the pre-rename era for proof-artifact byte-stability (EasyCrypt / Lean / Jasmin / dudect), and `MinInputLength=13` (1 version + 4 proof-len + 4 magic + 4 public-input-len). Gas schedule is `BaseVerifyGas=200,000 + PerByteGas=10`, sized to the CPU verify cost (5 ms to 10 ms per typical 30 kB proof). Soundness reduces to collision resistance of cSHAKE256 modeled as a random oracle; the parameter set used by the audited backend crate fixes the soundness error at $\leq 2^{-100}$. Completeness is 1.

Contents

1 Introduction

A post-quantum smart-contract platform must verify two different cryptographic objects on-chain:

1. **Signatures** (threshold or single-party) that authenticate *who* authored some commitment. Under Quasar this is the LP-218 [10] P3Q family at slot `0x012205`: kind-byte dispatch over Pulsar (FIPS-204 ML-DSA-65 threshold) [16], Corona (Ring-LWE threshold), Magnetar (FIPS-205 SLH-DSA threshold) [17].
2. **SNARK proofs** that authenticate *what* was executed – the correctness of an arbitrary off-chain computation with respect to a public-input vector. Under Quasar, this is LP-221: a STARK-FRI verifier at slot `0x012220`.

These are disjoint primitives. A Pulsar signature does not assert "the rollup batch executed correctly"; it asserts "the entity holding the threshold key co-signed this batch." A STARK proof does not assert "the sequencer was authorized"; it asserts "starting from `prev_root`, executing the batch yields `new_root`, and the public-input `batch_hash` commits to the right calldata." Both checks are required for a sound ZK-rollup – and they belong at separate precompile slots because they verify separate statements with separate parameter schedules, separate gas profiles, and separate audit boundaries.

1.1 Slot `0x012220`

LP-221 specifies a dedicated EVM precompile at slot:

```
0x0000000000000000000000000000000000012220
```

The slot is registered normatively in LP-300 master schema registry [12]. LP-221 carries the slot informatively for self-containment.

1.2 Distinction from LP-218 P3Q

The naming history is instructive. The Plonky3-fork STARK / FRI verifier originally registered at slot `0x012205` under the package name `p3q`, and the magic-header tag `"P3Q1"` was threaded through every proof artifact (EasyCrypt theory, Lean theorem, Jasmin source, dudect harness). Per Hanzo Crypto Suite § 5.2 [3], **P3Q canonically means "Post-Quantum Pulsar Proof"** – the on-chain unified PQ *threshold-signature* verifier dispatch (Pulsar / Corona / Magnetar). The STARK / FRI dispatch occupying slot `0x012205` under that name was an *aliasing error*.

The 2026-06-03 decomplection (Section 2) relocates the STARK-FRI verifier to its own dedicated slot at `0x012220` and leaves `0x012205` to LP-218 / LP-220 as the canonical P3Q signature dispatch [11].

Slot	LP	Verifier surface
`0x012205`	LP-218 / LP-220	PQ threshold **signatures** (Pulsar, Corona, Magnetar)
`0x012220`	**LP-221 (this LP)**	STARK **proofs** (Plonky3-fork FRI / cSHAKE256 / Goldilocks)

Table 1: The two slots serve disjoint use cases. A ZK-rollup typically calls *both* (signature at `0x012205`, proof at `0x012220`) per batch; an optimistic rollup or a custody attestation calls only one.

1.3 Why STARK-FRI for ZK rollups

The Quasar / Lux primary network needs an on-chain SNARK verifier for two reasons:

1. **ZK-rollup batch verification.** A Tier-3 ZK rollup (per LP-204 [8]) commits to Z-Chain by submitting a `RollupBatchTx` that carries both a Pulsar signature and a STARK proof. The Pulsar signature authenticates the *who* (sequencer or threshold-group identity); the STARK proof authenticates the *what* (correct execution of the batch under the rollup-VM's transition relation). Without an on-chain SNARK verifier, Z-Chain can only run optimistic rollups with fraud-proof challenge windows. With LP-221, Z-Chain accepts ZK-rollup batches at the cost of one precompile call per batch.
2. **PQ-safe SNARK substrate.** Most production SNARK verifiers (Groth16, PLONK over BN254 / BLS12-381) are pairing-based and break under a sufficiently large quantum adversary because pairing verification reduces to discrete log on a pairing-friendly elliptic curve. STARK proofs over a small prime field with collision-resistant hash commitments have *no algebraic structure for Shor*; their post-quantum security reduces to collision resistance of the underlying hash (here, cSHAKE256, modeled as a random oracle).

LP-221 therefore complements LP-218 (signatures) as the strict-PQ SNARK leg of the Quasar cryptographic suite. There is no classical-curve assumption anywhere in the LP-221 verifier.

1.4 Why FRI, cSHAKE256, Goldilocks

The three choices are deliberate:

- **FRI** [1, 2] is the canonical low-degree test for STARKs. Soundness reduces to a Reed-Solomon decoding bound; no trusted setup.
- **cSHAKE256** [14, 15] is the FIPS-202 / SP-800-185 customizable extendable-output function. Domain-separated Merkle commitments use cSHAKE256 rather than a Merkle-tree hash baked from a non-standard sponge. The standardization gates conformance and audit.
- **Goldilocks** ($p = 2^{64} - 2^{32} + 1$) is a 64-bit prime field [4] with fast modular arithmetic on AArch64 / x86_64, native register width, and a smooth multiplicative subgroup of order 2^{32}. Plonky3 [18] and RISC Zero converged on Goldilocks for the same reasons.

Combined: a verifier with ∼30 kB proof size (measured for typical Plonky3-fork batches), ∼5 ms to 10 ms CPU verify on a modern x86_64 core, and PQ security under the random-oracle model for cSHAKE256. Throughput on Blackwell sm_120 is projected at ∼50 µs per verify (LP-203 [7] GPU verify dispatch; not yet measured at LP-221 publish time).

1.5 Document roadmap

Section 2 documents the 2026-06-03 slot relocation from `0x012205` to `0x012220` and the Plonky3 fork rationale. Section 3 pins the wire format, gas schedule, and ABI. Section 4 catalogues the Go reference, the EasyCrypt theory, and the Jasmin status-code table. Section 5 states the security assumptions (soundness, knowledge soundness, completeness, PQ-safety). Section 6 describes the three-layer CT discipline (Jasmin static checker, Go inheritance, dudect empirical). Section 7 composes LP-221 with LP-218 (sibling PQ verifier), LP-0120 / LP-300 (slot pin), LP-203 (GPU dispatch), and LP-217 (cert profile modes).

2 Architectural Decision: Dedicated Slot for STARK-FRI

2.1 The 2026-06-03 slot relocation

Pre-rename state: the Plonky3-fork STARK / FRI verifier lived at slot `0x012205` under the package name `p3q`. The package contained the line *"P3Q strict-PQ STARK verifier"* and the magic-header tag `"P3Q1"` was threaded through every proof artifact (EasyCrypt, Lean, Jasmin, dudect).

Per Hanzo Crypto Suite § 5.2 [3] and the roadmap § B.11, **P3Q canonically means "Post-Quantum Pulsar Proof"**: the on-chain unified PQ *threshold-signature* verifier dispatch (Pulsar / Corona / Magnetar). The STARK / FRI dispatch occupying slot `0x012205` under the same name was an aliasing error: STARK proof verification and PQ signature verification are two different primitives serving disjoint use cases.

The 2026-06-03 decomplection makes this explicit by moving each concern to its own slot:

Surface	Pre-rename	Post-rename	Reason
Source path	`lux/precompile/p3q/`	`lux/precompile/starkfri/`	primitive name
Slot	`0x012205`	`0x012220`	disjoint surface
Go package	`package p3q`	`package starkfri`	value-typed naming
Address symbol	`ContractP3QVerifyAddress`	`ContractStarkFRIVerifyAddress`	matches package
Singleton type	`P3QVerifyPrecompile`	`StarkFRIVerifyPrecompile`	matches package
Config key	`p3qVerify`	`starkfriVerify`	matches package
Wire-bytes magic	`"P3Q1"`	`"P3Q1"` (preserved)	proof-artifact stability

Table 2: Surfaces touched in the 2026-06-03 rename. Slot moves; package and symbol names follow the package; the wire-bytes magic header is preserved verbatim because every proof artifact references it.

After the rename:

- `0x012205` is LP-218's unified PQ *signature* verifier (P3Q-Pulsar / P3Q-Corona / P3Q-Magnetar – kind-byte dispatched per LP-220).
- `0x012220` is LP-221's STARK-FRI *proof* verifier (this LP).

The two slots are orthogonal. There is no aliasing, no overlap, no shared dispatch table. Both can be called from a single ZK-rollup batch validator and the two costs sum cleanly: ∼50 000 gas for Pulsar verify + ∼500 000 gas for STARK verify ≈ 550 000 gas per ZK-rollup batch at typical parameters.

2.2 Why `"P3Q1"` survives the rename

The 4-byte tag `"P3Q1"` is a *wire-format byte string*, not a name claim. Renaming it would invalidate the byte-identical proof artifacts (EasyCrypt theories at admit-budget 0 / 0, Lean theorems, Jasmin source, dudect harness) that reference the header verbatim. A future v2 wire format MAY rename the magic to `"SFR1"` (STARK-FRI Rev 1) once the proof artifacts and the artifact-budget tracker (`scripts/checks/ec-admits.sh`) are refreshed in lockstep. Today, breaking proof-artifact byte equality for a cosmetic rename has no upside.

This is the *values, not places* discipline: the magic header is the value at the wire, not a label about which file it lives in; renaming the file does not rename the wire-bytes value, and the proof artifacts that reference the wire-bytes value stay byte-stable.

2.3 Lockstep updates across the codebase

The rename touched five downstream surfaces in the same atomic landing. These are recorded for self-containment; the normative surface remains the precompile package itself.

- `lux/geth/core/vm/lux_precompiles.go` – EVM wiring, registry entry updated to `starkfri.StarkFRIVerifyPrecompile` at `0x012220`.
- `lux/chains/evm/main.go` – chain import shim.
- `lux/evm/precompile/registry/registry.go` – precompile registry.
- `lux/crypto/p3q/ct/dudect/verify_ct.go` – dudect harness; cross-tree rename to `lux/crypto/starkfri/ct/dudect/` pending (see § 6).
- LP-218 §"Verifier reference" – inline reference to the STARK-FRI precompile now points at `0x012220` rather than the old `0x012205` co-location.

2.4 Why a Plonky3 fork

The reference backend at `lux/p3q/crates/p3q-verifier` is a Plonky3 [18] fork. The fork strips:

- KZG polynomial commitments (require pairings, classical-only).
- BN254 / BLS12-381 elliptic-curve backends.
- All non-strict-PQ FRI configurations (e.g. small-field Reed-Solomon parameter choices that don't meet the soundness target of $\leq 2^{-100}$ under the chosen blowup factor and query count).

Plonky3 upstream is dual-tracked: a strict-PQ profile (cSHAKE256 / Goldilocks / FRI only) and a classical profile (KZG / Poseidon / BN254). LP-221's fork pins the strict-PQ profile and removes the classical surface entirely. The two reasons:

1. **Audit surface.** An EVM precompile is a permanent reservation in block validation. Every line of code reachable from the precompile entry point is in the audit boundary. A dual-tracked upstream with both classical and PQ surfaces doubles the audit boundary without doubling the security guarantee – only the strict-PQ surface is reachable in production.
2. **Misuse resistance.** A future caller cannot accidentally instantiate the classical surface by passing the wrong configuration bytes; the classical surface does not exist in the binary. The wire-format version byte (`VersionV1=0x01`) selects *which strict-PQ profile* the verifier expects; it does not select between classical and PQ.

Tracking the Plonky3 upstream beyond this point is a *rebase-and-re-strip* discipline: each upstream release the fork rebases, re-applies the strip patches, re-runs the EasyCrypt admit budget check, and re-tests against the proof corpus. The fork maintainer is one person on the cryptography team; the rebase is not a routine release-engineering concern.

2.5 What this is not

It is worth being explicit about two design decisions LP-221 does *not* make:

1. **It is not a kind-byte-dispatched family.** Unlike LP-218 / LP-220 at slot `0x012205` where a single precompile dispatches over $\{0x01, 0x02, 0x03\}$ to three signature verifiers, LP-221 is a single-primitive precompile. The `VersionV1=0x01` version byte exists so a future LP can graft a v2 wire format (post-Plonky3-upstream-rotation or a recursive variant) without re-allocating the slot, but the version is not a primitive-family discriminator. The slot is reserved for STARK-FRI; if a fundamentally different SNARK family enters the Lux stack (a recursive HyperPlonk variant, say) it lands at its own slot inside the reserved range `0x012220..0x01222F`.
2. **It is not a backwards-compatible upgrade of a prior STARK precompile.** There is no pre-LP-221 STARK verifier deployment in the wild. LP-0120's historical row at `0x012205` for "Plonky3 PQ-only STARK" was a pre-genesis placeholder that never reached any network. The activation marker `2025-12-25T16:20:00-08:00` places the LP-221 ABI at network genesis of the new final Lux network. The pre-Quasar Edition Lux network (2020–2025) had no STARK-FRI precompile; there is nothing to migrate, no shim, no replay window.

2.6 Activation marker

```
activates:      2025-12-25T16:20:00-08:00
activates-unix: 1766708400
```

The STARK-FRI precompile at slot `0x012220` is callable from the genesis block of the new final Lux network.

3 Specification: Wire Format, Gas, ABI

3.1 Slot and config key

Attribute	Value
EVM precompile address	`0x0000000000000000000000000000000000012220`
Go address symbol	`ContractStarkFRIVerifyAddress`
Chain-config key	`starkfriVerify`
Go package	`starkfri`
Reference path	`lux/precompile/starkfri/`

Table 3: Slot and registration identifiers. The slot pin in LP-300 [12] is normative; this table is informative.

The chain-config key `starkfriVerify` is the unique key under which the upgrade-config block is keyed in the node's chain configuration JSON. The reference implementation registers under this key in `module.go` via the standard precompile registration init function.

3.2 Wire format

Single-call interface; no Solidity function selector. The wire bytes are:

```
[ 1 byte  ] version            -- 0x01 (VersionV1)
[ 4 bytes ] proof_len           -- big-endian uint32
[ proof_len bytes ] proof       -- must begin with MagicHeader "P3Q1"
[ 4 bytes ] pub_len             -- big-endian uint32
[ pub_len bytes ] public_inputs
```

Field	Size	Encoding	Constraint
`version`	1 byte	raw byte	MUST equal `0x01`
`proof_len`	4 bytes	big-endian `uint32`	MUST equal length of `proof`
`proof`	variable	opaque bytes	First 4 bytes MUST equal `"P3Q1"`
`pub_len`	4 bytes	big-endian `uint32`	MUST equal length of `public_inputs`
`public_inputs`	variable	opaque bytes	format dictated by circuit

Table 4: Wire-format field table. Big-endian for length fields matches Ethereum-mainnet conventions and the LP-218 P3Q ABI.

3.2.1 Minimum input length

$$\texttt{MinInputLength} = 1 + 4 + 4 + 4 = 13 \text{ bytes}$$

The thirteen bytes decompose as: 1 version byte + 4 proof-length bytes + 4 magic-header bytes (which live inside the `proof` blob but are structurally required) + 4 public-input-length bytes. Calls shorter than 13 B are rejected with `ErrInvalidInputLength`.

The Jasmin reference verifier checks `len(input) >= MinInputLength` as the first gate, before the version byte or magic-header check. This ordering is intentional: length checks are cheaper than byte-string equality on the magic header, and the dudect harness measures the dispatch path under all three rejection modes (short, wrong-version, wrong-magic) to empirically confirm timing uniformity.

3.2.2 Magic header

The 4-byte tag `MagicHeader = "P3Q1"` is preserved verbatim from the pre-rename era so EasyCrypt / Lean / Jasmin / dudect proof artifacts remain byte-identical against on-chain calldata. The header is a *wire-bytes tag, not a name claim*; renaming would invalidate the proof artifacts. A future v2 wire format may rename it to `"SFR1"` once the proof artifacts are refreshed and the artifact-budget tracker (`scripts/checks/ec-admits.sh`) is updated in lockstep. This is documented at `contract.go:46-52` in the reference implementation.

3.2.3 Version byte

Only `VersionV1 = 0x01` is accepted today. Versions `0x00` or $\geq$ `0x02` are rejected with `ErrInvalidVersion`. The version byte exists so a future LP can graft a v2 wire format (post-Plonky3-upstream-rotation or a recursive variant) without re-allocating the slot.

3.3 Gas schedule

$$\texttt{RequiredGas}(\texttt{input}) = \texttt{BaseVerifyGas} + |\texttt{input}| \cdot \texttt{PerByteGas} = 200{,}000 + |\texttt{input}| \cdot 10$$

Constant	Value	Units
`BaseVerifyGas`	200 000	gas
`PerByteGas`	10	gas / byte

Table 5: Gas constants per `contract.go:107-110`.

The 200 000 base reflects FRI verify cost being *logarithmic in circuit size*: on-chain we only see the serialized proof, so we charge a flat base plus per-byte for bandwidth and decode. At a typical 30 kB proof, total gas is $200{,}000 + 30{,}000 \cdot 10 = 500{,}000$ gas per verify – $\sim$24 verifies per 12 000 000-gas C-Chain block.

3.3.1 Upfront gas billing

Gas is charged upfront before any structural validation or backend dispatch (`contract.go:194-198`). The `oog_dominates` EasyCrypt lemma (`proofs/easycrypt/P3Q_Gas_Model.ec`) pins this: out-of-gas wins against every other failure mode. The lemma's statement is informally "for every input and every `gas_in` value, if `gas_in < RequiredGas(input)` then the precompile's return tuple is $(\texttt{nil}, 0, \texttt{ErrOutOfGas})$ regardless of any other field's contents."

This matters for soundness because it forecloses a gas-side channel: a caller cannot probe which validation stage rejected the input by arranging `gas_in` to be just-below the per-stage threshold, because there are no per-stage thresholds. There is one threshold, charged upfront, and either it is met (in which case validation runs) or it is not (in which case the call short-circuits with `ErrOutOfGas` before any byte of the input is parsed).

3.4 Return contract

This contract differs from the LP-218 `abiFalse`-on-failure convention because LP-221 reports *structural* failures via Go error returns. A Solidity caller will see the precompile call revert on any structural mismatch (length, version, magic) and on backend verification failure; success returns empty output (the proof either verifies or it doesn't, and the caller's contract treats the absence of revert as success).

The `ErrInvalidProof` sentinel is overloaded to cover both *wire-format* bad-magic and *cryptographic* backend rejection. A future LP may split these into two sentinels (`ErrBadMagic` vs

Outcome	Return
Verified	(nil, gasLeft, nil)
Insufficient gas	(nil, 0, contract.ErrOutOfGas)
Input shorter than MinInputLength, or internal length fields exceed remaining bytes	(nil, gasLeft, ErrInvalidInputLength)
version != 0x01	(nil, gasLeft, ErrInvalidVersion)
proof[0:4] != "P3Q1"	(nil, gasLeft, ErrInvalidProof)
No verifier registered via RegisterVerifier	(nil, gasLeft, ErrVerifierNotRegistered)
Backend returned (false, nil)	(nil, gasLeft, ErrInvalidProof)
Backend returned (_, err) for err != nil	(nil, gasLeft, err) – backend internal failure surfaces verbatim

Table 6: Return-tuple contract by outcome. Per contract.go:113-118, error sentinels are exported package symbols so Solidity callers (via revert-data decoding) and Go callers can pattern-match on them.

ErrBackendReject); today they share the same revert-data path because the caller's response is identical (reject the batch). Splitting them is a UX improvement, not a soundness improvement; it lands behind the version-byte gate when the wire format bumps.

3.5 Cross-reference to the registry

LP-300 §"Post-quantum precompile registry" is the master pin for slot 0x012220. The pre-LP-221 row in LP-300 read:

> 0x012220 - LP-218 - Placeholder for displaced Plonky3-fork STARK - pending dedicated LP allocation.

LP-221 closes the placeholder. LP-300's row should be amended on its next revision to:

> 0x012220 - LP-221 - STARK-FRI - Plonky3-fork FRI / cSHAKE256 / Goldilocks STARK verifier; magic header "P3Q1" preserved verbatim for proof-artifact stability - Draft - 2026-06-04.

LP-300 is normative; LP-221 is the dedicated LP that LP-300's placeholder row was waiting on.

4 Verifier: Go Reference, Jasmin Table, EasyCrypt Budget

4.1 The Go reference implementation

The reference implementation is single-source-of-truth at lux/precompile/starkfri/ [13].

4.1.1 Measured test result

```
$ go test -race -count=1 -v ./...
...
PASS
ok  github.com/luxfi/precompile/starkfri  1.376s
```

22 tests pass under go test -race -count=1 -v ./... at HEAD on main of github.com/luxfi/precompile. Measured 2026-06-04 on darwin/arm64; total wallclock 1.376 s including race detector. The race detector catches no data race; the structural-parsing path and the registered-verifier atomic load are concurrency-safe by construction.

Artifact	LOC	Purpose
`contract.go`	244	`StarkFRIVerifyPrecompile` struct; `Run()` does parse, structural pre-filter, dispatch
`module.go`	57	`init()` registers `starkfriVerify` config key + slot `0x012220`
`contract_test.go`	156	8 unit tests (address pin, gas formula, structural rejection, version, registered verifier round trip, missing-verifier surfaces typed error, wrong-version, backend-rejection surfaces typed error)
`contract_e2e_test.go`	∼440	14 tests covering 64 round-trip dispatches, structural rejection before backend, backend-failure surfacing, gas monotonicity, gas uniformity, OOG short-circuit, round-trip parser; 5 Go fuzz targets + native `FuzzStarkFRI_Dispatch`

Table 7: Reference artifact inventory. Total Go source under `lux/precompile/starkfri/` is ∼900 LOC.

4.1.2 Backend dispatch boundary

The Go layer is intentionally thin: ∼70 LOC of `Run()` body is structural parsing and pre-filter; the heavy lifting (FRI verify, cSHAKE256 Merkle openings, Goldilocks arithmetic) happens in the audited Rust verifier crate at `lux/p3q/crates/p3q-verifier`, reached through `RegisterVerifier` at node startup. The cgo bridge is out of scope of LP-221 (it is a node-build detail) and is gated by an external audit per the sign-off's GATE-4.

The atomic-load pattern on the verifier function pointer means a node operator can hot-swap the backend without restarting the precompile (e.g., to roll out a verifier patch in a coordinated upgrade window); concurrent calls observe either the pre-swap or post-swap function value, never a torn read.

4.2 Jasmin reference and status-code table

The Jasmin reference at `lux/precompile/starkfri/jasmin/verify.jazz` (14.2 kB) covers the structural pre-dispatch gate only: length checks, version byte, magic-header equality via the XOR-accumulator `ct_check_magic`. The FRI verify inner loop is delegated to the Rust verifier crate; Jasmin owns the boundary that EasyCrypt reasons about.

Status code	Symbol	Meaning
0	`STARKFRI_OK`	Structural gate passed; dispatch to backend
1	`STARKFRI_ERR_LENGTH`	`len(input) < MinInputLength`, or internal length fields overrun
2	`STARKFRI_ERR_VERSION`	`version != 0x01`
3	`STARKFRI_ERR_MAGIC`	`proof[0:4] != "P3Q1"` (XOR-accumulator zero check)
4	`STARKFRI_ERR_NO_VERIFIER`	no backend registered at call time

Table 8: Jasmin status codes returned by the structural gate. The Go layer maps these into Go error sentinels per Table 6 in Section 3.

4.2.1 CT annotation

The CT annotation in the Jasmin source is:

```
#[ct = "public * public * public * public -> public"]
```

All four arguments and the return value are tagged `public`. This is consistent with the threat model: STARK proofs and their public inputs are non-secret by construction. The CT annotation is load-bearing for soundness rather than confidentiality (see Section 6).

4.3 EasyCrypt admit budget: 0 / 0

The EasyCrypt theory under `lux/precompile/starkfri/proofs/easycrypt/` comprises four files and is pinned at admit budget 0 / 0:

File	Subject	Admits used	Admits budgeted
`P3Q_Verifier.ec`	accept-iff-backend-accept bridge	0	0
`P3Q_Gas_Model.ec`	`oog_dominates` lemma	0	0
`P3Q_CT.ec`	CT obligation ⇒ `backend_ct` axiom	0	0
`P3Q_Magic_Header.ec`	magic-header equality reduces to byte-string equality	0	0
Total		**0**	**0**

Table 9: EasyCrypt admit budget. Pinned at `scripts/checks/ec-admits.sh` which counts the literal token `admit` across all theory files and fails CI on any non-zero result.

The budget is enforced by a per-push CI check; any new `admit` token in any of the four files breaks the build. The zero-admit budget means every theorem in the theory either discharges to a Lean / EasyCrypt primitive, to a backend axiom explicitly named in the theory, or to a structural-Jasmin correspondence claim.

The key theorem is `accept_iff_backend_accept` in `P3Q_Verifier.ec`:

> *For every input `calldata`, every gas-in `gas_in` ≥ `RequiredGas(calldata)`, and every registered backend `B`: the precompile returns* (`output`, _, `nil`) *on calldata if and only if the backend `B` accepts* (`version=0x01`, `proof`, `pub`) *extracted from `calldata` per the wire-format parse.*

The theorem reduces precompile soundness to backend soundness; the backend's soundness is the standard STARK / FRI knowledge soundness under the random-oracle model.

4.4 Sign-off gates

The internal cryptographer sign-off (`CRYPTOGRAPHER-SIGN-OFF.md` in the precompile package) verdict is **APPROVED WITH GATES**. The four gates are:

1. **GATE-1.** EasyCrypt admit budget pinned at 0 / 0 in CI (**closed**; `scripts/checks/ec-admits.sh` enforces on every push).
2. **GATE-2.** Jasmin file passes `jasminc -checkCT` on every push (**closed**; integrated into `scripts/check-high-assurance.sh`).
3. **GATE-3.** Submission-grade dudect run (10^9 samples per stage on the structural pre-filter) (**open**; harness builds clean, smoke runs at 10^4 samples per push are *weak assertions, not measurements*; the production-grade run is gated on hardware scheduling).
4. **GATE-4.** External audit of the Rust verifier crate at `lux/p3q/crates/p3q-verifier` (**open**; audit procurement in progress).

LP-221 ships with GATE-1 and GATE-2 closed. GATE-3 and GATE-4 are prerequisites for moving LP-221 from Draft to Final; the slot allocation and ABI are stable across the Draft→Final transition, only the assurance posture changes.

5 Cryptographic Properties

5.1 Threat model

The verifier operates in the following threat model:

- **Adversary capability.** Probabilistic polynomial-time Turing machine. PQ adversary is a polynomial-time quantum Turing machine; the verifier's PQ claim is against this stronger adversary class.
- **Adversary access.** The adversary chooses calldata (proof bytes and public inputs) and submits it to a node's EVM. The adversary observes the precompile's return tuple but not the validator's internal state.
- **Honest verifier.** The validator runs the unmodified precompile + backend; we assume no fault injection on the validator side.
- **Random oracle for cSHAKE256.** cSHAKE256 is modeled as a random oracle in the security reduction. This is the standard assumption for FRI's Fiat-Shamir compilation; the modeled primitive is the cSHAKE256 customizable XOF as standardized in FIPS-202 / SP-800-185.

There are *no secret keys* on the verifier side. STARK proofs authenticate public statements; the verifier's secrets-side surface is empty. The CT discussion in Section 6 therefore concerns soundness leakage (which validation stage rejected) rather than confidentiality leakage (no confidentiality to leak).

5.2 Soundness

5.2.1 Knowledge soundness

Under the random-oracle model for cSHAKE256, FRI is a knowledge-sound interactive oracle proof [1, 2]. Knowledge soundness extracts a witness from any prover that convinces the verifier with probability above the soundness error.

Concretely, for the strict-PQ Plonky3-fork configuration used by the audited backend crate, the knowledge-soundness statement is:

> *There exists a polynomial-time extractor E^P such that for every PPT prover P that wins the FRI knowledge-soundness game with probability ϵ, the extractor outputs a witness for the underlying R1CS / Plonkish instance with probability $\geq \epsilon - 2^{-100}$.*

The 2^{-100} figure is the *soundness error*, derived from the FRI parameter set: query count q, blowup factor ρ, folding factor f, and field size $|\mathbb{F}|$. The Plonky3-fork parameter set is documented in `lux/p3q/crates/p3q-verifier/src/params.rs`; LP-221 pins the verifier-conformance, not the prover parameter schedule (the prover may choose stronger parameters but cannot choose weaker ones, because the verifier hard-codes its expected schedule).

5.2.2 Soundness error

Per the parameter schedule used by the audited backend crate, the soundness error is $\leq 2^{-100}$ in the random-oracle model. This is the conventional STARK target [2]: it bounds the probability that a cheating prover convinces the verifier of a false statement.

5.2.3 Completeness

Honest prover, honest verifier: the verifier accepts with probability 1. There is no statistical-zero-knowledge slack at the verify side; the prover may have ZK slack (depending on whether the Plonky3-fork hiding mode is enabled), but the verifier's acceptance probability on an honestly-generated proof is exactly 1.

5.2.4 Public-coin / Fiat-Shamir

FRI is public-coin [1]; the Fiat-Shamir transform with cSHAKE256 as the random oracle compiles it to a non-interactive argument. The transcript hashing domain-separates rounds; cSHAKE256's customization string carries the per-round domain tag.

5.3 Post-quantum assumption

The single cryptographic assumption is **collision resistance of cSHAKE256**, modeled as a random oracle in the Fiat-Shamir reduction. There is no discrete-log, no pairing, no factoring, no LWE, no SVP, no SIDH-style isogeny assumption.

cSHAKE256 [14, 15] inherits the SHA-3 / Keccak-$f[1600]$ design. The conjectured security strength is:

Adversary	Best attack	Conjectured cost
Classical	generic preimage / collision (birthday)	$\geq 2^{256}$ preimage; $\geq 2^{128}$ collision
Quantum	Grover preimage; BHT collision	$\geq 2^{128}$ preimage; $\geq 2^{85}$ collision

Table 10: cSHAKE256 security strength against classical and quantum adversaries at the 256-bit output length. Adequate for $\leq 2^{-100}$ soundness error at the FRI parameter choices the backend uses; the soundness reduction is loose by $\sim$30 bits of margin against the Grover-bounded quantum attacker, which is the standard PQ-STARK safety budget.

5.3.1 Why not BLAKE3 or Poseidon

LP-221 picks cSHAKE256 over alternatives for three reasons:

1. **Standardization.** cSHAKE256 is FIPS-202 / SP-800-185 [14, 15]. NIST standardization gates conformance test vectors, third-party crypto-library implementations, and audit checklists. BLAKE3 is widely deployed but not FIPS-standardized; Poseidon is SNARK-friendly but its collision-resistance under standard-model assumptions is the subject of ongoing analysis.
2. **Customization string.** cSHAKE256's customization string (S parameter) is purpose-built for domain separation between protocol contexts. The Plonky3-fork uses per-round customization strings to bind Fiat-Shamir transcripts; reusing the same primitive with different S values is cleaner than concatenating tag-prefixes onto a non-customizable hash function.
3. **Audit surface uniformity.** The Quasar stack uses Keccak-$f[1600]$ as the hash primitive at multiple layers (cSHAKE256 here, SLH-DSA inside Magnetar [17], EVM KECCAK256 at the precompile boundary). Co-locating the hash primitive reduces the audit surface and amortizes the side-channel hardening across multiple protocol layers.

5.4 Domain separation

The on-chain wire format carries a fixed leading `MagicHeader = "P3Q1"` that ties the precompile to the strict-PQ Plonky3 fork. Within the proof bytes, the Plonky3-fork verifier applies its own domain separation via cSHAKE256 customization strings on each Fiat-Shamir round; that domain separation is internal to the backend crate and is out of scope of LP-221's wire-format contract.

The two-layer domain separation (wire-bytes magic header at the EVM boundary + Fiat-Shamir customization inside the backend) is deliberate. The EVM boundary's magic header is a coarse type-discriminator: it answers "is this a STARK-FRI strict-PQ v1 proof or something else." The backend's customization strings are fine-grained: they answer "which round of which

proof is this hash invocation for." Conflating the two is an anti-pattern ("omit the magic header because the backend will catch it later") because it shifts a cheap, structural check into an expensive, cryptographic one.

5.5 What is NOT claimed

- **Zero-knowledge.** The Plonky3-fork verifier as deployed does NOT carry a zero-knowledge property at the LP-221 boundary. Public inputs (`prev_root`, `new_root`, `batch_hash`) are explicitly public. A future LP can graft a ZK-extension by activating Plonky3's hiding commitment mode and bumping the version byte. Today, LP-221 verifies computational integrity, not zero-knowledge proof-of-knowledge.
- **Universal setup.** Not applicable; FRI is transparent (no trusted setup of any kind). The Plonky3 fork inherits this property; LP-221 does not introduce any setup-requiring component.
- **Recursion.** The Plonky3-fork verifier as exposed at LP-221 does not recurse over itself on-chain; recursion happens off-chain in the prover and surfaces to the EVM as a single STARK proof. A future recursive variant would land at a separate version byte (and likely separate magic header) under the same slot.
- **Quantitative GPU verify performance.** LP-221 does NOT claim measured GPU throughput. The CPU verify cost of 5 ms to 10 ms on a modern x86_64 core is the measured number; the projected ∼50 µs per verify on Blackwell sm_120 is a projection from LP-203 kernel micro-benchmarks [7], not a measurement. The gas schedule is sized to the CPU-class cost.

6 Constant-Time Discipline

6.1 Three layers

LP-221 maintains constant-time guarantees through three orthogonal layers, ordered by decreasing strength:

1. **Jasmin static checker.** The Jasmin reference at `lux/precompile/starkfri/jasmin/verify.jazz` (14.2 kB) is annotated with a CT type: `#[ct = "public * public * public * public -> public"]`. The annotation is statically checkable via `jasminc -checkCT verify.jazz`. The CT check is part of `scripts/check-high-assurance.sh` and runs on every push. The Jasmin file covers the structural pre-dispatch gate only (length checks, version byte, magic-header equality via the XOR-accumulator `ct_check_magic`); the FRI verify inner loop is delegated to the Rust verifier crate.
2. **Go EVM precompile inheritance.** The Go body in `contract.go` is shaped to be CT-friendly: no secret-dependent branches; `Run()` conditions on *public* calldata lengths and the *public* version byte only; the magic-header comparison is a byte-string equality on *public* wire bytes. The Go layer is NOT formally CT-verified (Go lacks a libjade-style CT checker) but inherits the same byte-layout discipline as the Jasmin reference. **LP-221 does not claim the Go layer itself is constant-time as a primitive**; it claims only the byte-flow shape matches the CT-checkable Jasmin reference.
3. **Empirical CT via dudect.** The harness lives at `lux/crypto/p3q/ct/dudect/` (per the sign-off; relocation to `lux/crypto/starkfri/ct/dudect/` is pending the cross-tree rename). Builds clean on arm64 + x86_64; the submission-grade 10^9-sample run is gated under GATE-3 of the sign-off and has not yet been published.

Layer	Strength	Status
1. Jasmin `-checkCT`	Static, formal	Closed (GATE-2)
2. Go inheritance	Inherited byte-shape	Asserted; not formal
3. dudect empirical	Empirical, weak-vs-strong by sample size	Open (GATE-3); 10^9-sample run pending

Table 11: Three-layer constant-time discipline. Layer 1 is the strongest claim; layer 3 is the empirical confirmation; layer 2 is the inheritance bridge between them.

6.2 GATE-3 dudect requirement

The dudect harness's README explicitly documents that per-push smoke runs ($\sim 10^4$ samples) are a **weak assertion, not a measurement**. LP-221 makes no production-grade CT claim until GATE-3 closes. The closure criterion is:

- Sample count: $\geq 10^9$ per stage (short-input, wrong-version, wrong-magic, valid-pass-through).
- Hardware: representative validator-class CPU (current validator deployment standard is AMD EPYC 7763 or Intel Xeon Platinum 8480+; ARM validators run Ampere Altra or Apple M-series with their respective hard-CT properties).
- Result tag: t-statistic across all stages remains ≤ 4.5 (dudect's default conservative threshold) for the full sample budget.
- Published artifact: a signed CSV plus an EasyCrypt-checkable assertion that the measured timing distribution is statistically indistinguishable across the four stages.

GATE-3 is open and scheduled. LP-221 ships with the harness plumbed but the empirical measurement pending. The architectural decision to claim CT at the precompile boundary is independent of GATE-3 closure: GATE-3 *confirms* the static CT claim under realistic compiler output, but the soundness argument does not depend on it.

6.3 Why CT matters for a public proof

STARK proofs and their public inputs are *non-secret by construction*; side-channel leakage of "the proof verifies" vs "the proof doesn't verify" is not a confidentiality violation. However, CT is required for **soundness**: a timing oracle over the four failure modes (magic-mismatch, length-mismatch, wrong-version, no-verifier-registered) lets an adversarial caller probe the chain's structural pre-filter and craft calldata that exploits the boundary between gas billing and structural rejection.

The Jasmin reference and the Go shape both ensure that pre-filter outcomes are not data-dependent on values an adversary can manipulate per call. Specifically:

- Length checks use unsigned comparison on *public* lengths (the calldata-length field is observable on-chain; no secret is involved).
- Version-byte check is equality on a single *public* byte against a compile-time constant.
- Magic-header check uses an XOR-accumulator `ct_check_magic` that processes all four bytes in fixed time independently of which byte (if any) differs from the expected `"P3Q1"` tag.
- No-verifier-registered check is an atomic-load of a function pointer plus a non-secret nil-check.

None of these introduces a timing channel that an adversary can amplify into a soundness break. The dudect run under GATE-3 is the empirical confirmation that the Go compiler's output (which is the production-deployed binary, not the Jasmin reference) preserves this property under realistic optimization passes.

6.4 Backend CT discharge

The cgo handoff to the Rust verifier crate is the external CT obligation. The EasyCrypt theory's `lemmas/P3Q_CT.ec` reduces the precompile-level CT obligation to a backend axiom `backend_ct`. That axiom is discharged by:

- the Jasmin file's `#[ct = ...]` annotation (statically checkable with `jasminc -checkCT`), and
- the audited Rust verifier crate (`#![forbid(unsafe_code)]`, panic-free by contract), pending the external review under GATE-4.

6.5 Compiler output preservation

A subtlety worth flagging: the Go compiler may perform branch prediction, dead-code elimination, or instruction reordering that preserves the byte-shape claim but alters the timing distribution. The dudect run under GATE-3 measures the production binary, not the Go source, and is the authoritative confirmation that the compiler's output preserves CT.

The Jasmin reference is *source-level* CT-verified; the empirical layer measures *binary-level* CT. The two together bracket the assurance: if the Jasmin source is CT and the production binary's dudect distribution is statistically indistinguishable across failure modes, then either the Go compiler preserved CT or the dudect harness has insufficient power to detect the violation. Either way, an adversary's exploitation budget against the timing channel is bounded above by the dudect power (sample count · measurement precision), which the 10^9-sample budget pushes well below cryptographic significance.

7 Composition With Sibling LPs

7.1 LP-218 P3Q at slot `0x012205`

LP-218 [10] defines `0x012205` as a single precompile that takes a leading kind byte ($0x01$ Pulsar, $0x02$ Corona, $0x03$ Magnetar) and verifies a PQ **threshold signature**. LP-220 [11] extends LP-218 with the Corona and Magnetar kind specifications on the same slot. LP-221 (this LP) defines `0x012220` as a single precompile that verifies a STARK **proof**. The two are composed at the rollup layer:

- LP-218 `RollupBatchTx` (schema `0xE8`) carries both a `pulsar_sig` field and a `ZKProofOff` field.
- A **ZK-rollup** variant carries both: the Pulsar signature authenticates the sequencer's commit, and the STARK proof authenticates correct execution of the batch under the rollup-VM's transition relation.
- An **optimistic-rollup** variant carries only the Pulsar signature; `ZKProofOff = 0` and the integrity model is fraud-proof challenges (LP-218 schema `0xE9 RollupChallengeTx`).

Z-Chain block validation under the ZK-rollup variant calls **both precompiles**: `0x012205` for the signature, `0x012220` for the proof. Both must accept. No nested call. Gas budget is the sum: Pulsar verify ∼50 000 + STARK verify ∼500 000 ≈ 550 000 gas per ZK-rollup batch under typical parameters.

7.1.1 Sample ZK-rollup validator (Solidity)

```
function validateZKBatch(
    bytes32 prevRoot,
    bytes32 newRoot,
    bytes32 batchHash,
    bytes calldata pulsarSig,
    bytes calldata groupPk,
    bytes calldata starkProof,
```

```
    bytes calldata pubInputs
) external returns (bool) {
    bytes32 h = sha256(abi.encodePacked(prevRoot, newRoot, batchHash));

    // Signature leg: WHO authored this batch
    if (!P3Q.verify(0x01, pulsarSig, h, groupPk)) return false;

    // Proof leg: WHAT was executed
    if (!STARKFRI.verify(VersionV1, starkProof, pubInputs)) return false;

    return true;
}
```

Two precompiles, two slots, two checks. The Solidity caller does not need to know that they live at different slots; the import shims (`P3Q.verify` at `0x012205`, `STARKFRI.verify` at `0x012220`) hide the slot allocation behind named function calls.

7.2 LP-0120 PQ precompile band

LP-0120 owns the `0x012201..0x012208` PQ-signature / KEM band. LP-221 sits one nibble above at `0x012220`, **chosen so the strict-PQ STARK family can grow its own sub-range (`0x012220..0x01222F`) without recolliding with the PQ-signature block**. Future SNARK-family allocations (recursive STARK, hash-based zkVM variants, etc.) land inside `0x012220..0x01222F` per LP-300 master registry amendments.

Slot range	Owner	Family
`0x012201..0x012208`	LP-0120	PQ signatures + KEMs (raw primitives)
`0x012205`	LP-218 / LP-220	P3Q signature dispatch (rollup-batch wrapper)
`0x012220..0x01222F`	LP-221+	STARK SNARK family (reserved)
`0x012220`	LP-221 (this LP)	STARK-FRI v1

Table 12: PQ precompile slot map. LP-0120 / LP-218 / LP-221 are deliberately spaced for sub-range growth.

7.3 LP-300 master schema registry

LP-300 [12] is normative for the slot pin. LP-221 carries the slot informatively. Any future amendment to LP-221's wire format, gas schedule, or version byte must be registered in LP-300 first.

The LP-300 row for `0x012220` is closed by this LP. Per Section 3 §"Cross-reference to the registry", the post-LP-221 row should read:

```
0x012220 - LP-221 - STARK-FRI - Plonky3-fork FRI / cSHAKE256 /
Goldilocks STARK verifier; magic header "P3Q1" preserved verbatim
for proof-artifact stability - Draft - 2026-06-04.
```

7.4 LP-4800 and LP-4820 (related primitives)

LP-4800 and LP-4820 are the broader strict-PQ SNARK substrate LPs (parameter schedules, transcript bindings, and recursive-proof extensions). LP-221 implements the EVM-callable verifier surface for the LP-4800 baseline; recursive variants from LP-4820 will land at separate slots inside the reserved `0x012220..0x01222F` range with their own version bytes or dedicated LPs.

The split is deliberate: LP-221 is normative for the on-chain ABI and gas; LP-4800 / LP-4820 are normative for the prover-side parameter discipline. A future LP-221 amendment

that graft a new prover parameter schedule must coordinate with the corresponding LP-4800 amendment to keep the prover/verifier parameter pair in lockstep.

7.5 LP-203 GPU-native verify

LP-203 [7] specifies the Blackwell sm_120 batched-verify dispatch path. A STARK batch verify kernel for LP-221 is projected at `lux/accel/cuda/starkfri.cu` (not yet implemented at LP-221 publish time). The projected per-verify cost on Blackwell sm_120 is ∼50 µs amortized across a batch of ∼64 proofs; the gas schedule above is sized to the CPU-class verify cost (5 ms to 10 ms), so GPU acceleration is a node-operator latency improvement, not a gas reduction. LP-221 does NOT claim measured GPU throughput; that lands when the kernel ships.

The LP-203 dispatch shim (`lux/accel/c_api.h`) will gain `lux_starkfri_verify_batch` as one of its entry points once the kernel ships; the precompile's `RegisterVerifier` hook accepts the GPU-backed verifier function as a drop-in replacement for the CPU verifier, with the hot-swap pattern preserved (atomic load on the verifier function pointer).

7.6 LP-217 cert profile modes

LP-221 is orthogonal to the QuasarCert ladder (PQ-off, PQ-fast, PQ-strict, PQ-heavy) [9]. The STARK verifier is invoked inside EVM execution, not inside consensus cert formation. Cert profile mode affects how LP-221's precompile call is **finalized** (which signature families validate the containing block) but does not affect the precompile's verify cost or correctness.

LP-217 mode	Block cert leg set	LP-221 STARK verify
`PQ-off`	BLS only	callable; result trusted under classical adversary
`PQ-fast`	BLS + P3Q($0x01$ Pulsar)	callable; result trusted under PQ adversary
`PQ-strict`	BLS + P3Q($0x01$) + P3Q($0x02$ Corona)	callable; result trusted under cross-family PQ adversary
`PQ-heavy`	BLS + P3Q($0x01$) + P3Q($0x02$) + P3Q($0x03$ Magnetar)	callable; result trusted under three-family-independent PQ adversary

Table 13: LP-217 cert profile mode affects block finalization (the signature legs that validate the containing block), not the STARK-FRI precompile's verify path. The verify path is the same across all four modes; the cert mode determines what *else* must verify to finalize the block carrying the STARK verify call.

7.7 LP-300 registry entry

The canonical LP-300 row for LP-221, as it should appear in the next LP-300 amendment:

```
slot:    0x012220
LP:      LP-221
name:    STARK-FRI
abi:     STARKFRI.verify(uint8 version, bytes proof, bytes pubInputs) -> bool
gas:     200000 + len(input) * 10
version: 0x01 (VersionV1)
magic:   "P3Q1"
backend: Plonky3 fork (cSHAKE256 / Goldilocks / FRI strict-PQ)
```

status: Draft
created: 2026-06-04

7.8 Cross-reference table

LP	Relationship to LP-221
LP-078 [6]	EVM precompile suite – broad allocation policy; LP-221 sits in the strict-PQ band per that policy
LP-0120 [5]	Quasar mainnet defaults – the historical 0x012205 STARK row vacated to 0x012220 by this LP
LP-203 [7]	GPU-native verify – projected Blackwell batched-verify kernel for STARK-FRI
LP-204 [8]	Network of blockchains – Tier-3 ZK rollups consume the LP-221 precompile per batch
LP-217 [9]	Cert profile modes – orthogonal; affects block finalization, not precompile semantics
LP-218 [10]	P3Q signature dispatch at sibling slot 0x012205; defines the rollup pattern that consumes LP-221 in the ZK variant
LP-220 [11]	P3Q Corona + Magnetar kinds at sibling slot 0x012205; specifies the other two PQ-signature kinds that compose with LP-221 STARK verify
LP-300 [12]	Master schema registry; normative slot pin; placeholder row for 0x012220 closed by this LP

Table 14: LP-221 composition references.

7.9 Backwards compatibility

None. The pre-Quasar Edition Lux network (2020–2025) had no STARK-FRI precompile and is a separate network out of scope. The genesis state of the new final Lux network (activation 2025-12-25T16:20:00-08:00) ships the LP-221 ABI at slot 0x012220 as the only ABI; no shim exists for any pre-LP-221 STARK verifier (there was no such verifier on any running network).

7.10 Future work

- **GATE-3 dudect closure** (Section 6): 10^9-sample dudect run on the structural pre-filter, producing a signed CSV plus an EasyCrypt-checkable assertion of statistical indistinguishability.
- **GATE-4 external audit**: third-party review of the Rust verifier crate at lux/p3q/crates/p3q-verifier.
- **GPU batch dispatch**: implement lux/accel/cuda/starkfri.cu and expose lux_starkfri_verify_batch via c_api.h. Once landed, the precompile's verifier function pointer can be hot-swapped to the GPU-backed implementation for validators with Blackwell-class GPUs.
- **Cross-tree CT-harness rename**: relocate lux/crypto/p3q/ct/dudect/ to lux/crypto/starkfri/ct/dudect/ in lockstep with the EasyCrypt theory file rename. Defer until the proof-artifact byte-stability requirement is reassessed.
- **Recursive STARK variant**: a future LP at slot 0x012221 (or a higher version byte at 0x012220) for the recursive-proof variant once the prover-side parameter discipline lands in LP-4820.

7.11 Activation marker

```
LP-221 Activation
-----------------
Activates:       2025-12-25T16:20:00-08:00
Activates-Unix:  1766708400
Status:          Draft
Slot:            0x0000000000000000000000000000000000012220
Config key:      starkfriVerify
Wire version:    0x01 (VersionV1)
Magic header:    "P3Q1" (preserved verbatim from pre-rename era)
Gas:             BaseVerifyGas (200,000) + len(input) * 10
Reference:       lux/precompile/starkfri/
Tests passing:   22 / 22 under `go test -race -count=1 -v ./...`
                 (measured 2026-06-04 on darwin/arm64, 1.376s)
Master pin:      LP-300 "Post-quantum precompile registry" row 292
                 (placeholder closed by this LP)
EasyCrypt:       4 theory files, admit budget 0 / 0
Jasmin:          verify.jazz CT-checked via `jasminc -checkCT`
Sign-off:        APPROVED WITH GATES (GATE-1, GATE-2 closed;
                 GATE-3 dudect $10^9$, GATE-4 external audit open)
```

Per LP-300 / LP-217 convention, LP-221 activates at the genesis of the new final Lux network. The pre-Quasar Edition Lux network (2020–2025) had no STARK-FRI precompile; there is nothing to migrate, no backwards-compat layer, no replay window.

References

[1] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast Reed-Solomon interactive oracle proofs of proximity. In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, pages 14:1–14:17. Schloss Dagstuhl, 2018.

[2] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity, 2018. IACR ePrint 2018/046. https://eprint.iacr.org/2018/046.

[3] Hanzo AI. Hanzo crypto suite, 2026. Section 5.2: P3Q canonical definition ("Post-Quantum Pulsar Proof"); Section B.11: STARK / P3Q rename rationale.

[4] Daniel Lubarov, Brendan Farmer, et al. The Goldilocks 64-bit prime field for STARK provers, 2022. Polygon Zero technical note; $p = 2^{64} - 2^{32} + 1$.

[5] Lux Industries. Lp-0120: Quasar mainnet defaults (precompile slot registry), 2026.

[6] Lux Industries. Lp-078: EVM precompile suite, 2026.

[7] Lux Industries. Lp-203: GPU-native verify on Grace Blackwell, 2026.

[8] Lux Industries. Lp-204: Network of blockchains, 2026.

[9] Lux Industries. Lp-217: Cert profile modes, 2026.

[10] Lux Industries. Lp-218: ZAP-native pq rollups via p3q, 2026.

[11] Lux Industries. Lp-220: P3q corona + magnetar kinds, 2026.

[12] Lux Industries. Lp-300: The lux schema registry, 2026.

[13] Lux Industries. Stark-fri verifier precompile reference implementation, 2026. `lux/precompile/starkfri/`; 22 tests passing under `go test -race -count=1 -v ./...` (measured 2026-06-04 on darwin/arm64, 1.376s).

[14] NIST. FIPS 202: SHA-3 standard: Permutation-based hash and extendable-output functions, August 2015. https://csrc.nist.gov/pubs/fips/202/final.

[15] NIST. SP 800-185: SHA-3 derived functions: cSHAKE, KMAC, TupleHash, and ParallelHash, December 2016. https://csrc.nist.gov/pubs/sp/800/185/final.

[16] NIST. FIPS 204: Module-lattice-based digital signature standard (ML-DSA), 2024. https://csrc.nist.gov/pubs/fips/204/final.

[17] NIST. FIPS 205: Stateless hash-based digital signature standard (SLH-DSA), 2024. https://csrc.nist.gov/pubs/fips/205/final.

[18] Polygon Zero / Plonky3 contributors. Plonky3: A toolkit for implementing polynomial-iop-based zksnarks, 2024. https://github.com/Plonky3/Plonky3.