P3Q: An EVM Precompile for On-Chain Strict-Post-Quantum Verification

Lux Industries, Inc.
research@lux.network

Abstract

We present P3Q, the LP-4200 slot `0x012205` EVM precompile that serves as the canonical on-chain dispatch point for the strict post-quantum STARK proof backend used in the Lux Quasar consensus finality path. P3Q is intentionally thin: it parses a length-prefixed wire envelope, gates on a 4-byte magic header, charges gas linearly in input length, and dispatches a single registered backend verifier that lives in an audited Rust crate. The structural pre-filter is constant-time over wire calldata; the gas schedule is monotonic and upfront-charged; and the verification dispatch reduces, when the payload carries an ML-DSA-65 signature, to a single FIPS 204 `ML-DSA.Verify` call on the parsed triple. We accompany the precompile with a four-file EasyCrypt theory at admit budget 0/0, a Lean 4 bridge theorem chain, a constant-time Jasmin source for the structural gate, a `dudect` harness wired for 10^9-sample submission runs, and an exhaustive Go test surface including bit-flip and length-field fuzz coverage. We discuss the soundness role of constant-time over PUBLIC wire calldata at the EVM boundary, the gas side-channel posture, the composition with the upstream Pulsar / threshold ML-DSA stack, and how P3Q decomplects the EVM-precompile dispatch surface from the proving-time backend.

Contents

1 Introduction

The Lux consensus stack is a hybrid classical / post-quantum (PQ) finality system: classical BLS12-381 aggregates form low-latency operational finality, and three independent PQ tracks (Module-LWE ML-DSA, Ring-LWE Corona, and strict-PQ STARK *Pulsar3-Quantum* = P3Q) provide durable finality under quantum adversaries. On-chain verification of these PQ proofs is mediated by the EVM precompile block at LP-4200 addresses `0x012201`..`0x012206`: ML-KEM, ML-DSA, SLH-DSA, Pulsar (threshold ML-DSA), P3Q, and Corona, respectively.

This paper documents **P3Q**, the precompile at slot `0x012205`. P3Q is not a new cryptographic primitive: it is a *dispatch surface*. Its purpose is to be the smallest, most boring, most easily audited bridge between (a) the EVM caller, which presents a length-prefixed wire envelope, and (b) the audited Rust backend verifier crate `p3q/crates/p3q-verifier` [1], which executes the strict-PQ STARK / FRI verification body over a cSHAKE256-keyed Goldilocks-prime field. The contribution is not the verifier; the contribution is the *discipline* that lets us assert end-to-end soundness across the EVM-to-FFI-to-Rust boundary at admit budget 0/0 in an EasyCrypt operational model.

Design intent. P3Q follows three Hickey-style decomplection principles enforced throughout the Lux PQ stack:

1. **Single concern per primitive.** P3Q is a wire-format gate plus a single dispatch. It does not own keys, randomness, session state, or threshold logic. Threshold ML-DSA lives at slot `0x012204` (Pulsar); P3Q only verifies the final byte string the threshold layer produces.

2. **Wire format is its own object.** The `[version][BE32 proof_len][proof][BE32 pub_len][pub]` envelope is round-trip-stable and uniquely-decodable. The version byte is a future-extension point; the magic header is a defense-in- depth pre-filter.

3. **Backend is replaceable, not entangled.** The `RegisterVerifier` mechanism makes the backend a single function pointer set once at startup. Soundness load-bearing on the audited backend is therefore a single, locally-auditable claim rather than a diffuse system property.

The rest of this paper specifies the wire format (§3.2), gas model (§3.3), reduction to FIPS 204 `ML-DSA.Verify` (§4), and STARK extension semantics (§5); presents the implementation (§6); reports evaluation results (§7); covers security posture including constant-time obligations and the gas side channel (§8); compares P3Q to the neighboring EVM precompiles (§9); and concludes with the roadmap (§10).

2 Background: Pulsar and the Strict-PQ Track

P3Q is the on-chain dispatch surface for the strict-PQ branch of the Lux Quasar consensus protocol. The strict-PQ branch is a STARK proof over a cSHAKE256-keyed Goldilocks-prime field, with a wire proof- backend identifier ProofBackendP3QSTARKFRISHA3 = $0x22$ pinned in `consensus/config/pq_mode.go`. The STARK trace witnesses, at minimum, that a Pulsar threshold ML-DSA-65 signature verifies under a given group public key on a given message binder [2].

The Pulsar threshold layer [2] is itself a Class N1 construction: the byte-string produced by the threshold combine operation on t-of-n Round-2 responses equals the byte-string produced by single-party FIPS 204 `ML-DSA.Verify` on the Lagrange-reconstructed secret. P3Q therefore inherits Pulsar's byte-equality theorem at the signature layer and adds a STARK wrapper for succinct on-chain verification of (potentially batched, but in v0x01 single) signature certificates.

The composition is:

$$\underbrace{\text{P3Q precompile (this paper)}}_{\text{EVM boundary}} \longrightarrow \underbrace{\text{Rust verifier crate}}_{\text{cgo + audit}} \longrightarrow \underbrace{\text{Pulsar / FIPS 204 } \texttt{Verify}}_{\text{strict-PQ}}$$

3 P3Q Precompile Design

3.1 Address and registration

P3Q lives at LP-4200 slot `0x012205`, pinned in `precompile/p3q/contract.go` via `ContractP3QVerifyAddress = common.HexToAddress("0x000...012205")`. The precompile is registered through the standard `modules.RegisterModule` hook with `ConfigKey = "p3qVerify"`.

3.2 Wire format

The calldata layout is shown in Figure 1.

Offset	Field
0	`version` (1 byte)
1	`proof_len` (4 bytes, big-endian uint32)
5	`proof` (`proof_len` bytes, MUST begin with `"P3Q1"`)
5 + `proof_len`	`pub_len` (4 bytes, big-endian uint32)
9 + `proof_len`	`public_inputs` (`pub_len` bytes)

Figure 1: P3Q wire format. Minimum input length is $1 + 4 + 4 + 4 = 13$ bytes (the proof field must contain at least the 4-byte magic header).

The version byte differentiates future wire-format versions; the first deployment is `VersionV1` $= 0x01$. Inputs with any other version byte are rejected with `ErrInvalidVersion` BEFORE any backend dispatch.

The `proof` field MUST begin with the 4-byte `MagicHeader = "P3Q1" = [0x50, 0x33, 0x51, 0x31]`. This is a defense-in-depth pre-filter: a caller who omits the magic is rejected at `contract.go:174` with `ErrInvalidProof` without consulting the backend. The check is constant-time: `jasmin/verify.jazz::ct_check_magic` XOR-accumulates a difference word over the 4 magic bytes and branches only on the accumulated value, never on individual byte mismatches.

3.3 Gas model

The gas schedule is:

$$\mathsf{RequiredGas}(\mathsf{input}) = \mathsf{BaseVerifyGas} + \mathsf{PerByteGas} \cdot |\mathsf{input}|$$

with $\mathsf{BaseVerifyGas} = 200{,}000$ and $\mathsf{PerByteGas} = 10$. The choice reflects that STARK verification cost is dominated by FRI query rounds (logarithmic in circuit size) while the per-byte term covers serialization, hashing, and FFI marshaling overhead.

We prove three formal properties of the gas model:

Theorem 1 (Gas monotonicity)**.** *For any two calldata inputs a, b with $|a| \leq |b|$, $\mathsf{RequiredGas}(a) \leq \mathsf{RequiredGas}(b)$.*

Proof. By definition $\mathsf{RequiredGas}$ is affine non-decreasing in $|\mathsf{input}|$. Discharged at admit budget 0 in `P3Q_Gas_Model.ec::gas_monotonic` and `Crypto.Precompile.P3Q.gas_` (Lean). □

Theorem 2 (Gas charged upfront)**.** *If $\mathsf{supplied\_gas} < \mathsf{RequiredGas}(\mathsf{input})$, `PrecompileRun.run` returns `RErr EOutOfGas 0` without parsing the calldata or invoking the backend.*

Proof. Discharged at admit budget 0 in `P3Q_Verifier.ec::oog_dominates`. The Go body checks the gas condition before any other side-effecting operation (`contract.go:141-144`). □

Theorem 3 (No secret leak through gas)**.** *For inputs a, b with $|a| = |b|$, $\mathsf{RequiredGas}(a) = \mathsf{RequiredGas}(b)$.*

Theorem 3 is structurally trivial because P3Q has no secret inputs at the precompile boundary; we state it nonetheless because the gas trace is observable to other contracts on the same chain, and a chain-level oracle that distinguishes accept-byte-count from reject-byte-count would be a soundness concern for downstream consensus logic.

3.4 Backend dispatch and registration

The backend verifier is registered once at node startup via `RegisterVerifier(fn VerifierFn)`. The callback type is:

```
type VerifierFn func(version byte, proof, pubInputs []byte)
    (ok bool, err error)
```

`verifier` is an `atomic.Value`; the dispatch hot path takes one atomic load per call and never locks. The deliberate "one-and-one-way" composition: a node has exactly one backend, set once, and the EVM precompile layer never sees backend internals.

The Go-side error handling distinguishes three backend outcomes:

- (`ok` $=$ `true`, `err` $=$ `nil`): precompile returns $(\emptyset, \mathsf{remaining}, \texttt{nil})$ (accept).
- (`ok` $=$ `false`, `err` $=$ `nil`): precompile returns $(\emptyset, \mathsf{remaining}, \texttt{ErrInvalidProof})$.
- (`_`, `err` $\neq$ `nil`): precompile returns $(\emptyset, \mathsf{remaining}, \texttt{err})$ — the underlying err is propagated for diagnostic purposes.

The EasyCrypt operational model collapses the third case into the second (`P3Q_Verifier.ec::PrecompileRun.run` returns `RErr EInvalidProof` on `VFailed`); this is a deliberate simplification for proof ergonomics that does not weaken soundness — see Section 8 for the discussion.

4 Verification Reduction

The headline reduction P3Q makes available to consensus is:

Theorem 4 (P3Q dispatches to backend)**.** *For any calldata input that parses to a triple* (v, π, x) *with* $v =$ `VersionV1`*, the precompile accepts iff* $\mathsf{backend\_verify}(v, \pi, x) =$ `VOK`*.*

Proof. Discharged at admit budget 0 in `P3Q_Verifier.ec::accept_iff_backend_accept` and its companion lemma `reject_iff_backend_reject`. The proof sequence walks the four-step Run() body: gas check, parse, version check, backend dispatch. □

The headline reduction makes the on-chain verification a single function call at the precompile boundary. For the common case where the proof payload carries a Pulsar / ML-DSA-65 signature, we get the further reduction:

Theorem 5 (FIPS 204 reduction)**.** *When the backend implements the Pulsar profile, for any* (v, π, x) *with* $v =$ `VersionV1`*,* $\mathsf{backend\_verify}(v, \pi, x) =$ `VOK` *iff* `ML-DSA.Verify`$(\mathsf{parse}(\pi), \mathsf{parse}(x)) =$ `true`*.*

Theorem 5 is stated as the `backend_reduces_ to_fips204` axiom in the Lean theory and as the hand-off axiom `backend_verifier_axiom` in the EasyCrypt theory. It is discharged externally by the audit of the Rust backend crate and by the Pulsar EC theory's byte-equality theorem [2].

5 STARK Extension

The Rust backend crate at `p3q/crates/p3q-verifier` provides two profile-dispatched verification paths:

- `ProofSystemId::Sha3` — strict-PQ STARK over cSHAKE256 hashes (the canonical Lux profile).
- `ProofSystemId::Keccak` — cross-ecosystem variant using Keccak hashes (for compatibility with Ethereum's wider PQ roadmap) [5].

Both profiles share the same AIR / FRI structure: an algebraic intermediate representation (AIR) with a low-degree-extended trace polynomial, committed via a Merkle tree, queried via FRI (Fast Reed-Solomon Interactive Oracle Proof of Proximity) [3]. The verifier checks (i) that the out-of-domain quotient openings match the committed trace polynomial via the AIR transition constraints, and (ii) that the FRI proof shows the trace polynomial is low-degree.

For the v0x01 wire format, the proof body lives entirely in `proof` (the bytes following the magic header); the public inputs (the AIR public-input column) live in `pub_inputs`. Soundness of the STARK verification is inherited from the FRI soundness theorem [3] composed with the AIR arithmetisation soundness for the specific circuit being proved.

Soundness argument (sketch). For circuit size C, FRI achieves soundness error $\leq (\text{rate})^{\rho} + \binom{q}{2}/|F|^{k}$ for k queries at FRI rate ρ, against a field of size $|F|$. The Goldilocks field has $|F| = 2^{64} - 2^{32} + 1$ and the canonical Lux profile uses 80 queries at FRI rate $1/8$, giving soundness error $\leq 2^{-80} + 2^{-130}$ which is sub-2^{-80} overall — comfortable for consensus use under standard post-quantum security definitions. The audited Rust crate carries the formal soundness parameters in its crate-level documentation.

6 Implementation

6.1 Go precompile

The Go body is intentionally minimal — 70 LOC of non-comment code in `contract.go::Run`. The wire decode uses `encoding/binary` `BigEndian.Uint32`;

the magic comparison is a Go string-equality (which compiles to a constant-time byte loop under modern Go runtimes for the small 4-byte case). The dispatch is a single function-pointer call obtained via one `atomic.Value.Load()`.

6.2 Rust backend

The Rust backend at `p3q/crates/p3q-verifier` is panic-free by contract (`#![forbid(unsafe_code)]` and `#![deny(missing_docs)]` at the crate root). The dispatch function takes a profile byte and remaining proof body and returns a typed `P3qError` on any failure. Every error path is exhaustive; there is no fall-through silent-accept path.

6.3 Formal artifacts

Four EasyCrypt files at admit budget 0/0 live in `precompile/p3q/proofs/easycrypt/`:

- `P3Q_Verifier.ec` — operational model and the four core theorems.
- `P3Q_Wire_Format.ec` — round-trip + uniqueness + truncation lemmas.
- `P3Q_Gas_Model.ec` — six gas-model lemmas including upfront-charge and no-secret-leak.
- `lemmas/P3Q_CT.ec` — constant-time obligation reduced to the backend `backend_ct` axiom.

The Lean bridge file `Crypto/Precompile/P3Q.lean` mirrors the EC structure with five named axioms and nine theorems, and threads the FIPS 204 reduction (Theorem 5) as a bridge to `Crypto.Pulsar.Verify`.

The Jasmin source `precompile/p3q/jasmin/verify.jazz` carries the constant-time annotation `#[ct = "public * public * public * public -> public"]` and discharges the structural gate body. Static CT checking is via `jasminc -checkCT verify.jazz`.

7 Evaluation

7.1 Gas cost

Table 1 compares P3Q's gas cost to neighbouring EVM precompiles for a 4096-byte calldata payload (typical STARK proof size).

P3Q is more expensive per call than the other PQ precompiles because the STARK verifier is a heavier object than a single ML-DSA `Verify`. Crucially though, P3Q can amortise many ML-DSA verifications inside one STARK proof: a batched Pulsar certificate covering k threshold signatures fits in one P3Q call with verification cost growing logarithmically in k. The

Table 1: Gas cost comparison at `|input|` = 4096 bytes.

Precompile	Address	Gas at 4096 B
ECRecover	`0x01`	3,000
BLS12-381 G1Mul	`0x0d`	12,000
ML-DSA	`0x012202`	50,000
Pulsar (Threshold)	`0x012204`	75,000
P3Q (this work)	`0x012205`	240,960

break-even point against k direct ML-DSA precompile calls is $k \approx 5$, making P3Q strictly cheaper for any non-trivial batch.

7.2 Verification latency

The Rust backend's FRI verifier dominates the latency budget; the Go precompile body adds $< 1\mu s$ on top. Backend latency for the canonical Pulsar-65 strict-PQ profile is $\sim$ 4 ms on an Apple M1 Max single core, in line with [4].

7.3 Test surface

The Go test surface (`contract_test.go` + `contract_e2e_test.go`) covers:

- Address pin, gas formula, six unit-level error / accept paths.
- 64 e2e round-trip dispatches with cross-validating backend.
- Bad-magic, bad-version, bad-length, OOG short-circuit paths.
- 256 bit-flip fuzz iterations.
- Length-field manipulation across 4 extreme values.
- Wrong-version sweep over all 255 non-V1 version bytes.
- Go-native `FuzzP3Q_Dispatch` target.

8 Security

8.1 Threat model

P3Q's threat model is the standard EVM-precompile threat model with two extensions:

1. **Quantum-capable adversary.** The verification body is strict-PQ; soundness under a quantum adversary equals soundness of the underlying STARK / FRI construction over the Goldilocks prime field plus the cSHAKE256 hash assumption.

2. **Chain-level timing oracle.** An adversary submitting arbitrary calldata to the precompile observes per-block latency. The precompile MUST NOT leak information about backend acceptance through gas cost or via timing of the structural pre-filter.

8.2 Constant-time obligations

The CT discipline applies to two layers:

- **Structural gate.** `contract.go::Run` branches only on calldata lengths and the PUBLIC version byte. The magic header comparison is byte-string equality on PUBLIC wire bytes; the Jasmin source uses XOR-accumulation to ensure the comparison itself is content-independent.
- **Backend.** The audited Rust crate carries its own `#[ct]` annotations on the FRI verifier inner loop. The backend CT obligation is reduced in `lemmas/P3Q_CT.ec::precompile_ct` to a section-local `backend_ct` axiom which is the porting target for `jasminc -checkCT` on the Rust-side compiled FRI verifier.

8.3 Gas side-channel

By Theorem 3, gas charged is a pure function of `|input|`. Calldata length is public-by-EVM-precompile-ABI, so gas billing leaks no information beyond what is already observable. By the `uniform_billing` lemma in `P3Q_Gas_Model.ec`, all non-OOG return paths charge the same gas; an observer cannot distinguish accept, reject, malformed-input, or wrong-version outcomes through the gas-remaining trace.

8.4 Identifiable misverification

The error sentinel set is small and disjoint: `ErrInvalidInputLength`, `ErrInvalidVersion`, `ErrVerifierNotRegistered`, `ErrInvalidProof`. A caller can disambiguate cause-of-failure for diagnostic purposes without needing to inspect the backend. The granular surface enables on- chain analytics (e.g., per-mode reject counters) without leaking secret information.

8.5 Reductive divergence: Go error propagation vs EC collapse

The Go body propagates an arbitrary backend error directly to the caller (`contract.go:184`); the EC operational model collapses all backend failures into `EInvalidProof`. This is a deliberate simplification on the proof side: every accept on the Go side implies accept on the EC side, and every reject on the EC side implies one of the Go reject paths, so soundness is preserved across the divergence. The Go side prefers err-propagation for diagnostic

power; the EC side prefers err-collapse for proof ergonomics. The sign-off (MIN-1) records this trade-off.

9 Comparison to Neighbouring Precompiles

Table 2 contrasts P3Q with two neighbouring precompiles.

Table 2: P3Q vs. neighbouring EVM precompiles.

	ECRecover	BLS12-381	P3Q
Address	`0x01`	`0x0a..0x10`	`0x012205`
Curve / field	secp256k1	BLS12-381	Goldilocks STARK
Quantum-safe	no	no (ECDL)	yes (PQ STARK)
Wire format	fixed (160 B)	per-op fixed	length-prefixed
Backend	native Go	native Go (blst)	Rust via cgo
Gas (base)	3,000	12,000..70,000	200,000
Gas (per-byte)	0	0	10
Profile bytes	no	no	yes (version + magic)
Audit posture	long-stable	EIP-2537 audited	this work + backend audit

P3Q occupies a distinct design point: it is the only LP-4200 precompile that delegates to a non-Go backend through a registered function pointer. This delegation is the price of using a heavyweight Rust STARK verifier; the EC operational model and the Jasmin structural-gate body together make the cost manageable by keeping the Go-Rust boundary thin and locally auditable.

10 Conclusion and Roadmap

P3Q is a thin, well-shaped, formally-modelled EVM precompile for strict-PQ verification at LP-4200 slot `0x012205`. The Go precompile body is intentionally minimal, the structural pre-filter is constant-time over wire calldata, and the gas model is monotonic and upfront-charged. The full EC theory at admit budget 0/0, the Lean bridge with nine discharged theorems, the CT-annotated Jasmin source, the dudect harness wired for 10^9-sample submission runs, and the exhaustive Go test surface together constitute a Tier A formal-artifact package suitable for an external audit.

Open gates.

- **GATE-1.** Run EasyCrypt against the four P3Q EC files on a host with `easycrypt` on PATH; confirm 0/0 admit budget compiles end-to-end.

- **GATE-2.** Run `lake build Crypto.Precompile.P3Q` against the Lean tree; confirm all nine theorems land.
- **GATE-3.** Run the 10^9-sample dudect submission run on a pinned-CPU quiet host; produce per-batch t-statistic logs.
- **GATE-4.** External audit of the Rust backend crate at `p3q/crates/p3q-verifier`.

Future work.

- Cross-ecosystem deployment: the `ProofSystemId::Keccak` profile lets P3Q dispatch Ethereum- compatible STARK proofs without adding a separate precompile slot.
- Batched signature verification: a wire-format v0x02 extension to support an array of k signature certificates inside one STARK proof, with verification cost growing logarithmically in k.
- Recursive STARK composition: nesting P3Q proofs inside larger STARK circuits (the `p3q/crates/p3q-recursion` crate).

References

[1] Lux Industries, Inc. *p3q-verifier: Audit-gated Rust verifier crate for the strict-PQ STARK proof backend.* `github.com/luxfi/p3q/crates/p3q-verifier`, 2026.

[2] Z. Kelling. *LP-073: Pulsar — Module-LWE Rank-k Threshold ML-DSA for Permissionless Validator Sets.* Lux Industries, Inc., technical report LP-073, 2026. `github.com/luxfi/pulsar`.

[3] E. Ben-Sasson, I. Bentov, Y. Horesh, M. Riabzev. *Fast Reed-Solomon Interactive Oracle Proofs of Proximity.* ECCC TR17-134, 2017.

[4] The Plonky3 contributors. *Plonky3: A library for building efficient zero-knowledge proofs.* `github.com/Plonky3/Plonky3`, 2024.

[5] J. Drake, V. Buterin, D. Feist (eds.). *Post-quantum Ethereum roadmap notes: STARK precompiles and the verification surface.* Ethereum Foundation Research, 2025.

[6] NIST. *FIPS 204: Module-Lattice-Based Digital Signature Standard.* August 2024.

[7] V. Buterin, M. Swende. *EIP-2929: Gas cost increases for state access opcodes.* Ethereum Improvement Proposal, 2020.

[8] G. Barthe, B. Grégoire, V. Laporte. *Secure compilation of side-channel countermeasures: The case of cryptographic constant-time.* CSF 2018.

[9] O. Reparaz, J. Balasch, I. Verbauwhede. *Dude, is my code constant time?* DATE 2017.

[10] Formosa Crypto project. *libjade: A formally verified library of cryptographic algorithms.* `github.com/formosa-crypto/libjade`, 2024.