LUX_STRICT_E2E_PQ: End-to-End Post-Quantum Architecture for the Lux Network

Zach Kelling

Lux Industries

2026

Abstract

Most "post-quantum blockchain" work today PQ-hardens *consensus* only: a threshold signature over block headers replaces BLS, and the rest of the stack (transaction authorisation, contract permits, peer handshakes, bridge admission, randomness, finality binding, contract auth profiles) continues to use ECDSA, Keccak, BN254 pairings, and KZG trusted setups. A Shor-capable adversary defeats this stack at every layer the threshold signature does not touch.

This paper specifies LUX_STRICT_E2E_PQ, the locked profile under which *every* byte that gates a state transition on Lux is bound to a FIPS-203 / FIPS-204 / FIPS-205 primitive or a SHA3-family transcript. The profile is a single `ChainSecurityProfile` (`luxfi/consensus@v1.23.6`, `config/security_profile.go:244`) with eleven enum-or-flag fields, all of which are zero-value-invalid; the profile carries a SHA3-384 digest of every other field (`ComputeHash`, `security_profile.go:835`) and that digest is pinned in genesis. Booting on a chain whose profile does not match the pinned digest fails closed — there is no permissive default.

We specify the fourteen surfaces the profile covers. On signed-transaction authorisation: a 17-field `TxAuthEnvelope` (`protocol/auth/tx_envelope.go:108`) whose signing digest is TupleHash256-bound under customization `LUX_TX_AUTH_V1`, replacing RLP-keccak for raw transactions. On smart-contract permits: an 11-field `PQPermit` (`protocol/auth/permit.go:70`) under `LUX_PQ_PERMIT_V1`, replacing EIP-2612. On the Z-Chain identity rollup: five operations (`Register` / `Rotate` / `Revoke` / `AuthorizeSession` / `CommitTxAuthBatch` / `CommitPermitBatch`) over a seven-root `ZRegistryRoots` aggregator. On finality: a 23-axis Q-Chain `ComputeRoundDigest` binding consumed by Pulsar, the Module-LWE threshold scheme that replaces BLS in the finality lane (`luxfi/pulsar`). On proofs: the p3q STARK / FRI stack over SHA3 with no KZG, no BN254, and no Groth16 wrappers (`luxfi/p3q@v0.0.1`, ten crates). On the P2P layer: an ML-KEM-768 / ML-KEM-1024 + ML-DSA-65 handshake with five cSHAKE customizations (`luxfi/node@dc906d281b`). On bridges: a refused-primitives `BridgeProfile`. On contract auth: a per-contract auth profile resolved through the `PQAuth` precompile at `0x012202`.

We close with a golden vector. Under the locked LUX_STRICT_E2E_PQ profile, `ChainSecurityProfile.ComputeHash` returns

`93efd103...323bfcd0`.

This 48-byte digest is the entire E2E security claim, byte-equal, end to end, across every Lux node that boots the genesis under this profile. A divergent node sees a different digest and refuses to start.

Honest claim. The crypto primitives in this paper — ML-DSA-65, ML-KEM-768, cSHAKE256, TupleHash256, FRI, Pulsar — are not new. What is new is the *composition*: a single locked profile that pins every surface a classical or post-quantum adversary could touch, with a single 48-byte digest binding the entire system together. The paper specifies the surface coverage at a level of detail sufficient to reproduce every signing transcript and every wire envelope byte-for-byte from `luxfi/consensus`, `luxfi/crypto`, `luxfi/pulsar`, `luxfi/p3q`, and `luxfi/node` canonicals.

The canonical claims/evidence table and the architectural commitments live in LP-105 (Lux Stack Lexicon) and HIP-0078 (Z-Chain registry); other Lux documents reference them by section, never duplicate them.

Contents

1 Introduction

1.1 The gap end-to-end PQ closes

The post-quantum blockchain literature has converged on a fast answer: replace the consensus signature. Threshold ML-DSA [9], Threshold Raccoon [12], lattice-based DKGs [21, 17], and the Lux Pulsar family [25] all PQ-harden the certificate that closes a block. None of them PQ-harden the bytes that authorise the transactions *inside* the block, the bytes that authorise the cross-chain handshakes around the block, the bytes that authorise the smart-contract spend, or the trusted-setup SRS that the proof system underneath assumes.

A Shor-capable adversary against the resulting stack does not need to attack the threshold lane. The adversary forges an ECDSA transaction signature, an EIP-2612 permit, a BLS bridge admission, a Diffie-Hellman handshake, or a Groth16 verification key. Any one of those primitives is sufficient to drain funds, hijack a contract, or admit a malicious block.

The NIST Multi-Party Threshold Cryptography report [33] names this layer (the "cryptographic surface") and lists fourteen categories of bytes that participate in chain state transitions. A proper PQ posture binds *every* category to a quantum-resistant primitive. Anything less is theatre.

LUX_STRICT_E2E_PQ is the locked profile under which every one of those fourteen surfaces is pinned to a FIPS 203 / 204 / 205 primitive [34, 35, 36] or a SHA3-family hash [30, 32], and that pinning is consumed at boot, at handshake, at transaction admission, at contract permit verification, at finality sealing, at bridge admission, and at every other gating point in the stack.

1.2 What the profile is, mechanically

The profile is a single Go struct, `ChainSecurityProfile` (`luxfi/consensus@v1.23.6`, `config/security_profil` with the following well-known forms:

- `ProfileLuxStrictPQ` — the locked production profile this paper specifies, golden hash `93efd103...323bfcd0`.
- `ProfileLuxFIPS` — equivalent posture, FIPS-only branding, distinct hash for audit attestation.
- `ProfileLuxPermissive` — testnet / devnet profile with classical-compat affordances; *never loaded on mainnet.*

The struct has eleven enum-or-flag fields plus aggregator lists; *every* enum field is zero-value-invalid. A profile produced by zero-initialising the struct fails `Validate` (`security_profile.go:Validate`) at boot. The profile's identity is its SHA3-384 digest under `ComputeHash` (`security_profile.go:835`), which is in turn TupleHash256-bound under customization `LUX_CHAIN_SECURITY_V1` so any byte flip changes the digest. The digest is what genesis pins. A divergent node sees a different digest and refuses to start.

1.3 What surface coverage means

Surface coverage is the proposition: every byte that the consensus engine, the execution engine, the bridge, the wallet, or a peer treats as authoritative for a state transition is bound, through transcripts and signatures, to a post-quantum primitive.

The fourteen surfaces, in dependence order:

1. **Profile boot.** `ChainSecurityProfile` loaded from genesis, `ComputeHash` matches pinned digest, banner emitted on boot (`luxfi/node@9df72a6f55`).
2. **Identity registration.** Validator pubkeys (FIPS 204 ML-DSA) are registered through Z-Chain `OpRegisterKey` with a proof of possession over the new key's own AccountID.
3. **Identity rotation.** `OpRotateKey` signed under the *outgoing* key, gated by the profile's `IdentitySchemeID`.

4. **Identity revocation.** `OpRevokeKey` signed under the revoked key plus a profile-pinned reason code.
5. **Session authorisation.** `OpAuthorizeSession` binds an ephemeral session pubkey under an account's long-term identity key, for bounded TTL and bounded capability scope.
6. **Transaction authorisation.** `TxAuthEnvelope` (`protocol/auth/tx_envelope.go:108`), 17 fields TupleHash256-bound under `LUX_TX_AUTH_V1`. Replaces RLP-keccak for raw transactions.
7. **Smart-contract permits.** `PQPermit` (`protocol/auth/permit.go:70`), 11 fields under `LUX_PQ_PERMIT_V1`. Replaces EIP-2612.
8. **Z-Chain commitment.** `ZRegistryRoots` seven-root aggregator (`registry/roots.go:40`) bound under `LUX_ZCHAIN_REGISTRY_ROOTS_V1`.
9. **Execution-side auth verification.** `VerifyAuthPassed` (`registry/auth_proof.go:118`) hot-path Merkle-inclusion-proof verifier consumed by the EVM, S-VM, and X-VM state transition functions.
10. **Finality binding.** Q-Chain `ComputeRoundDigest` binds 23 axes (canonical QBlock layout) consumed by the Pulsar finality certificate.
11. **Finality signature.** Pulsar-65 (`luxfi/pulsar`), Module-LWE threshold scheme over a 43-of-64 committee with identifiable abort. No BLS fallback.
12. **Proof verification.** p3q (`luxfi/p3q@v0.0.1`, ten crates on crates.io) — STARK / FRI over SHA3 Merkle commitments. No KZG, no BN254, no Groth16 wrapper.
13. **P2P transport.** ML-KEM-768 / ML-KEM-1024 KEM and ML-DSA-65 static identity (`luxfi/node@dc906d281b`), five cSHAKE customizations.
14. **Bridge admission.** `BridgeProfile` (`luxfi/bridge@v1.0.7`) refused-primitive list, `RPC` label `post_quantum_end_to_end` as the affirmative public statement.

This paper is the surface-by-surface specification of the fourteen claims.

1.4 Why not just classical compat

Two well-defined regimes coexist on Lux: the strict-PQ regime LUX_STRICT_E2E_PQ specifies, and a classical-compat regime under `ProfileLuxPermissive` that the testnet / devnet uses for migration testing and that retains an EIP-1559 / EIP-2930 / legacy raw-tx path. The classical-compat regime is the responsibility of the `ClassicalCompatRegistry` (`luxfi/node`, mempool admission), is gated by the profile bit `RequireTypedTxAuth`, and never appears on a locked production profile. Section 13 discusses what classical-compat means precisely; the short answer is that on LUX_STRICT_E2E_PQ the mempool refuses any transaction that is not a `TxAuthEnvelope`, full stop.

The classical-compat path exists because a phased migration is the only realistic posture for a chain that hosts billions of dollars of incumbent contracts. The strict-PQ path is what mainnet finalises under once the migration window closes (HIP-0078 §6 names the cut-over event). This paper specifies the strict-PQ path. The permissive path is not a fallback at the strict-PQ profile level; it is a distinct profile with its own digest.

1.5 Document map

Section 2 specifies the `ChainSecurityProfile` struct and the golden vector. Section 3 specifies the 17-field signing digest, the wire codec, and the verification gate. Section 4 specifies the 11-field permit, the contract-side auth profile, and the `PQAuth` precompile at `0x012202`. Section 5 specifies the Z-Chain identity rollup — the five ops, the seven-root aggregator, the Merkle-inclusion-proof hot-path verifier. Section 6 specifies the 23-axis QBlock binding and the Pulsar finality certificate. Section 7 specifies the Pulsar Module-LWE threshold scheme. Section 8 specifies the p3q STARK / FRI stack. Section 9 specifies the ML-KEM + ML-DSA P2P handshake. Section 10 specifies the C-Chain / P-Chain / X-Chain coverage. Section 11 specifies

the bridge `BridgeProfile`. Section 12 reports gas-cost and signature-size measurements. Section 13 states what is PQ versus what remains classical-compat. Section 14 locates this work in the surrounding literature.

The companion artefacts are LP-strict-e2e-pq (this paper), HIP-0077..0104 (specs), LP-168..179 (Lux mirrors), ZIP-0809..0820 (Zoo mirrors), and the strict-e2e-pq Lean 4 + TLA+ + Go property tree in `luxfi/proofs/strict-e2e-pq/` [28].

2 ChainSecurityProfile: the locked profile

2.1 Struct

The profile is a single Go struct (`luxfi/consensus@v1.23.6`, `config/security_profile.go:244`) with eleven enum-or-flag fields, two ordered-set aggregator lists, two numeric floors, and a SHA3-384 self-hash. We quote the production canonical:

Listing 1: ChainSecurityProfile fields. `luxfi/consensus@v1.23.6`, `config/security_profile.go:244`. Comments shortened for the paper; canonical comments live with the source.

```
type ChainSecurityProfile struct {
    ProfileID uint32

    ProfileName string
    ProfileHash [48]byte   // SHA3-384 over the canonical encoding

    HashSuiteID       HashSuiteID    // single mandatory hash family
    IdentitySchemeID  SigSchemeID    // validator identity
    FinalitySchemeID  SigSchemeID    // finality lane
    HighValueSchemeID SigSchemeID    // governance / treasury

    ProofPolicyID         ProofPolicyID
    AllowedProofBackends  []ProofBackendID
    AllowedProofFormats   []ProofFormatID

    MinSoundnessBits  uint16
    MinHashOutputBits uint16

    RequireTransparent      bool
    ForbidPairings          bool
    ForbidKZG               bool
    ForbidTrustedSetup      bool
    ForbidClassicalSNARKs   bool
    ForbidDevProofs         bool
    ForbidFallbacks         bool

    RequireTypedTxAuth      bool
    WalletSchemeID          WalletSchemeID
    ContractAuthSchemeID    ContractAuthID

    BridgeProfileID         BridgeProfileID
    DRBGSchemeID            DRBGSchemeID
}
```

2.2 Zero-value-invalid by construction

Every enum field is zero-value-invalid. The constants `HashSuiteNone`, `SigSchemeNone`, `ProofPolicyNone`, `WalletSchemeNone`, `ContractAuthNone`, `BridgeProfileNone`, `DRBGSchemeNone` all evaluate to zero and *none of them* are accepted by `Validate`. A profile that was zero-initialised — the canonical Go-default "empty struct" — fails validation *at the first enum field validation step*, before any byte of profile state is consulted. This closes the "forgot to load the profile" class of bug: there is no permissive default; an unloaded profile is a fatal profile.

2.3 HashSuiteID

`HashSuiteID` pins the single mandatory hash family. The strict-PQ profile sets

$$\texttt{HashSuiteID} = \texttt{HashSuiteSHA3-384-cSHAKE256}.$$

cSHAKE256 [32] is the customisable Keccak permutation underneath SP 800-185's TupleHash, ParallelHash, and KMAC constructions. Every signing transcript in the profile (Sections 3–5) binds its customization string into cSHAKE256, so two different protocols sharing inputs cannot produce colliding digests.

A profile change to a non-SHA3 hash family bumps the customization tag at every transcript site (`LUX_TX_AUTH_V1` → `LUX_TX_AUTH_V2`, etc.) and changes `ComputeHash`. That is a hard fork.

2.4 The three scheme fields

`IdentitySchemeID`, `FinalitySchemeID`, and `HighValueSchemeID` are three independent signature schemes:

- `IdentitySchemeID` signs validator registration, rotation, revocation, and session authorisation. Strict-PQ pins it to raw FIPS 204 ML-DSA-65 (`0x40`-block).
- `FinalitySchemeID` signs the Pulsar finality certificate. Strict-PQ pins it to Pulsar-65 (`0x50`-block).
- `HighValueSchemeID` signs governance checkpoints and treasury actions. Strict-PQ pins Pulsar-65 or Pulsar-87 — never Pulsar-44, which is devnet-only.

The three schemes are independent: an attacker who breaks ML-DSA-65 but not Pulsar loses identity but not finality, and vice-versa. The defence-in-depth posture is the point of giving the three lanes different schemes.

2.5 Proof posture

The proof-posture group is `ProofPolicyID`, `AllowedProofBackends`, `AllowedProofFormats`, `MinSoundnessBits`, `MinHashOutputBits`, and the six `Forbid*` flags.

Strict-PQ sets:

Table 1: Proof-posture pinning under LUX_STRICT_E2E_PQ.

Field	Strict-PQ value	Rationale
`ProofPolicyID`	`ProofPolicy128bitPQ`	128-bit PQ soundness floor
`AllowedProofBackends`	{`p3q`, `Plonky3-SHA3`}	STARK / FRI only
`AllowedProofFormats`	{`p3q-v0.1`}	expanded-pubkey form
`MinSoundnessBits`	128	matches MLWE assumption
`MinHashOutputBits`	384	SHA3-384 Merkle width
`RequireTransparent`	`true`	STARK/FRI transparent
`ForbidPairings`	`true`	no BN254 / BLS12-381
`ForbidKZG`	`true`	no polynomial trapdoors
`ForbidTrustedSetup`	`true`	no SRS
`ForbidClassicalSNARKs`	`true`	no Groth16 wrappers
`ForbidDevProofs`	`true`	refuse `0x70` block
`ForbidFallbacks`	`true`	fail-closed, no downgrades

The six `Forbid*` bits exist as defence in depth against the four classes of attack HIP-0078 §4 enumerates: (1) classical pairings smuggled through under a relaxed backend list; (2) KZG polynomial-commitment trapdoors; (3) trusted-setup SRS compromise; (4) classical-SNARK

wrappers around an otherwise PQ proof. Setting all four to `true` closes the four classes simultaneously. Setting all four is what *makes* the profile "strict".

2.6 Wallet / contract auth / bridge / DRBG

The remaining four enum fields cover the surfaces consensus does not see:

- `RequireTypedTxAuth` (`true`): every transaction admitted to the mempool MUST be a `TxAuthEnvelope`. Legacy RLP-keccak transactions are refused by the `ClassicalCompatRegistry` (Section 10).
- `WalletSchemeID`: the wallet-side signing scheme. Strict-PQ pins this to ML-DSA-65 (matching `IdentitySchemeID` on this profile; not coincidentally the wallet and the validator are the same scheme).
- `ContractAuthSchemeID`: the contract-side permit scheme. Strict-PQ pins ML-DSA-65; HIP-0083 lays the groundwork for ML-DSA-87 on high-value contracts.
- `BridgeProfileID`: the bridge admission profile. See Section 11.
- `DRBGSchemeID`: the deterministic random-bit-generator scheme for chain-level randomness sources. Strict-PQ pins `Hash_DRBG-SHA3-384` (NIST SP 800-90A [31]).

2.7 ComputeHash and the golden vector

`ComputeHash` (`security_profile.go:835`) is the canonical self-hash:

Listing 2: `ComputeHash` construction sketch. Real code at `security_profile.go:835` is byte-faithful; shown here for the paper.

```
func (p *ChainSecurityProfile) ComputeHash() ([48]byte, error) {
    if err := p.Validate(); err != nil {
        return [48]byte{}, err
    }
    parts := [][]byte{
        []byte("Lux/ChainSecurityProfile/v1"),
        u32BE(uint32(p.ProfileID)),
        []byte(p.ProfileName),
        []byte{byte(p.HashSuiteID)},
        []byte{byte(p.IdentitySchemeID)},
        []byte{byte(p.FinalitySchemeID)},
        []byte{byte(p.HighValueSchemeID)},
        []byte{byte(p.ProofPolicyID)},
        canonicaliseBackends(p.AllowedProofBackends),
        canonicaliseFormats(p.AllowedProofFormats),
        u16BE(p.MinSoundnessBits),
        u16BE(p.MinHashOutputBits),
        []byte{boolByte(p.RequireTransparent)},
        []byte{boolByte(p.ForbidPairings)},
        []byte{boolByte(p.ForbidKZG)},
        []byte{boolByte(p.ForbidTrustedSetup)},
        []byte{boolByte(p.ForbidClassicalSNARKs)},
        []byte{boolByte(p.ForbidDevProofs)},
        []byte{boolByte(p.ForbidFallbacks)},
        []byte{boolByte(p.RequireTypedTxAuth)},
        []byte{byte(p.WalletSchemeID)},
        []byte{byte(p.ContractAuthSchemeID)},
        []byte{byte(p.BridgeProfileID)},
        []byte{byte(p.DRBGSchemeID)},
    }
    return tupleHash48(parts, "LUX_CHAIN_SECURITY_V1"), nil
}
```

`Validate` runs first, so a malformed profile never reaches the hash. The list of parts is canonicalised in declaration order. The customization `LUX_CHAIN_SECURITY_V1` pins the layout: re-ordering, dropping, or adding any part bumps the customization to `V2` and is a hard fork.

For the production strict-PQ profile, the golden vector is:

`93efd103...323bfcd0.`

This 48-byte digest is the entire E2E security claim, byte-equal, across every Lux node that boots under LUX_STRICT_E2E_PQ. The Go test `TestChainSecurityProfile_GoldenHash` (`security_profile_test.go`) asserts this digest equals the production canonical at commit `12097034`. The strict-e2e-pq Lean 4 module (`StrictE2EPQ.lean`, see Section 14) re-states the golden vector as a Lean-level proposition; the strict-e2e-pq TLA+ module (`StrictPQProfile.tla`) consumes it as a model parameter.

2.8 Genesis pinning and boot

`ChainSecurityProfile` loads from genesis at `luxfi/node@9df72a6f55` during node startup. The boot path:

1. Read the profile manifest from the genesis snapshot.
2. Compute the hash. If `ComputeHash` returns an error, abort.
3. Compare against the pinned digest. If different, abort with a `PROFILE_HASH_MISMATCH` fatal.
4. Emit the `SECURITY PROFILE` banner with the profile ID, the profile name, and the digest's hex prefix.
5. Wire the profile into every subsystem that consults it (`auth`, `zchain`, `qchain`, `pulsar`, `p3q`, `net`, `bridge`).

The banner is the canonical operator-visible affirmation that the strict profile is loaded. The hex prefix is the first eight bytes (`93efd103` on production); the operator confirms by visual diff against a printed reference card. The banner is the only place in the boot where a human sees the profile identity; everywhere else, the profile is consumed by digest.

3 TxAuthEnvelope: a 17-field signed transaction

3.1 What it replaces

Every classical Ethereum-family transaction lineage — legacy raw RLP, EIP-2930 access list, EIP-1559 dynamic fee, EIP-4844 blob — shares the same authorisation primitive: ECDSA over keccak256 of an RLP-encoded payload. The payload format changes across tx types but the authorisation primitive is fixed. Both are classical: ECDSA falls to Shor, keccak is fine but the domain-separation gaps in RLP encoding admit cross-version replays [47, 14].

A strict-PQ chain needs a transaction-authorisation primitive that (1) signs under ML-DSA-65 or another FIPS-204 scheme, (2) binds every security-relevant field via TupleHash256 so a single byte flip on any field breaks the signature, (3) carries the profile identity on the wire so cross-profile replays are impossible, and (4) survives lazy-deserialiser bugs by refusing to admit a transaction whose decoded fields disagree with its decoded digest.

`TxAuthEnvelope` (`luxfi/consensus@v1.23.6`, `protocol/auth/tx_envelope.go:108`) is that primitive.

3.2 The 17 fields

Table 2: TxAuthEnvelope field list. Source: `protocol/auth/tx_envelope.go:108`.

#	Field	Role
1	`Version`	envelope version (uint16). $v = 1$ on strict-PQ.
2	`ProfileID`	profile under which signed.
3	`ChainID`	destination chain id.
4	`NetworkID`	destination network id (mainnet / testnet).
5	`AccountID`	48-byte signer account identifier.
6	`Nonce`	uint64 monotonic per-account nonce.
7	`ExpiryHeight`	uint64 chain height after which envelope is invalid.
8	`WalletSchemeID`	signature scheme (matches `profile.WalletSchemeID`).
9	`HashSuiteID`	hash suite (matches `profile.HashSuiteID`).
10	`FeePayer`	48-byte account paying gas.
11	`GasLimit`	uint64 gas allowance.
12	`MaxFee`	32-byte EIP-1559-style fee cap.
13	`CallRoot`	32-byte hash of the call payload.
14	`AccessListRoot`	32-byte hash of the access list.
15	`ZIdentityRoot`	32-byte Z-Chain identity root binding.
16	`AccountStateRoot`	32-byte expected account state root at submit.
17	`PublicKeyRef`	32-byte SHA3-256 of the signer's full ML-DSA pubkey.

The 17 fields are the security-relevant inputs to the signing transcript. The sole non-transcript field is `Signature`, which carries the 3309-byte ML-DSA-65 signature itself (variable-length under other schemes).

The choice of 17 fields is the smallest set that closes *every* replay class the protocol team enumerated in HIP-0085 §3:

- **Cross-chain replay** is closed by `ChainID` and `NetworkID`.
- **Cross-profile replay** is closed by `ProfileID`.
- **Cross-version replay** is closed by `Version`.
- **Cross-account replay** is closed by `AccountID` and `PublicKeyRef`.
- **Cross-fee-payer replay** is closed by `FeePayer`, `GasLimit`, `MaxFee`.
- **Stale state replay** is closed by `ExpiryHeight`, `AccountStateRoot`, and `ZIdentityRoot`.
- **Cross-permission-set replay** is closed by `AccessListRoot`.
- **Same-account same-state different-call replay** is closed by `CallRoot`.
- **Cross-scheme replay** is closed by `WalletSchemeID` and `HashSuiteID`.

Dropping any field would re-open a replay class. The 17-field minimum is audit-driven.

3.3 SigningDigest

`SigningDigest` (`tx_envelope.go:148`) is what the wallet signs:

Listing 3: `SigningDigest` construction. Source: `protocol/auth/tx_envelope.go:148`. Hash family is cSHAKE256 under `LUX_TX_AUTH_V1`, independent of `HashSuiteID` which is bound as data into the transcript.

```
func (e *TxAuthEnvelope) SigningDigest() [48]byte {
    parts := [][]byte{
        []byte(txAuthProtocolTag),       // "Lux/TxAuthEnvelope/v1"
        u16BE(e.Version),
        {byte(e.ProfileID)},
        u32BE(e.ChainID),
        u32BE(e.NetworkID),
        e.AccountID[:],
        u64BE(e.Nonce),
        u64BE(e.ExpiryHeight),
        {byte(e.WalletSchemeID)},
        {byte(e.HashSuiteID)},
        e.FeePayer[:],
        u64BE(e.GasLimit),
        e.MaxFee[:],
        e.CallRoot[:],
        e.AccessListRoot[:],
        e.ZIdentityRoot[:],
        e.AccountStateRoot[:],
        e.PublicKeyRef[:],
    }
    return tupleHash48(parts, txAuthSigningCustomization)
}
```

The hash family is cSHAKE256, output truncated to 48 bytes; the customization string is `LUX_TX_AUTH_V1`. `HashSuiteID` is bound *as data* into the transcript — it is not the hash family of the transcript itself. This means a profile change to `HashSuiteID` bumps the customization (`LUX_TX_AUTH_V1` → `V2`) at the same time it changes the data byte. Both changes are observable in the digest.

3.4 Verification gate

`VerifyTxAuthEnvelope` (`protocol/auth/tx_envelope.go`) is the profile-gated verifier. The order matters; we list it explicitly because the order is the security-relevant claim:

1. **Structural.** Envelope non-nil, profile non-nil, `Version` supported, every field bytes-of-the-right-width.
2. **Profile/id match.** `envelope.ProfileID` equals `profile.ProfileID`.
3. **Profile/scheme match.** `envelope.WalletSchemeID` equals `profile.WalletSchemeID`.
4. **Profile/suite match.** `envelope.HashSuiteID` equals `profile.HashSuiteID`.
5. **Profile typed-auth gate.** `profile.RequireTypedTxAuth` is `true`; if false this code path is not reached at all.
6. **Z-Chain binding.** `ZIdentityRoot` matches the current Z-Chain identity root for this epoch (look-up against `ZRegistryRoots.IdentityRoot`, Section 5).
7. **Account state binding.** `AccountStateRoot` matches the current account state root.
8. **Account / pubkey binding.** The resolved pubkey hashes to `PublicKeyRef`, and the pubkey is on the active key list for `AccountID` as of `ZIdentityRoot`.
9. **Expiry.** Current chain height $\leq$ `ExpiryHeight`.
10. **Nonce.** `envelope.Nonce` equals `account.Nonce` (no fast-forward, no replay).
11. **Signature.** ML-DSA-65 verify of `Signature` over `SigningDigest()` under the resolved pubkey, FIPS 204 §5.2 ctx set to a per-profile constant.

Steps 2–5 reject before the expensive signature verification runs. Step 6–8 reject before a stale-state replay can land. Step 11 is the only step that runs the lattice signature primitive, which is the only PQ-expensive step.

3.5 Comparison to EIP-7932

EIP-7932 [14] proposes a signature registry for EVM transactions that lets new schemes plug in at the transaction-type byte. The EIP is structurally compatible with this paper's goals — it would allow a chain to admit ML-DSA-65 transactions alongside ECDSA transactions — but it falls short in two ways:

- EIP-7932 does not standardise the *transcript*. Each scheme declares its own signing digest. Two schemes sharing inputs can produce different digests because the scheme-byte appears in the wrapper but not in the inner transcript. TupleHash256 with a single customization closes that gap.
- EIP-7932 does not standardise the profile binding. A transaction signed under scheme A on chain X can be replayed as a scheme-A transaction on chain Y if both chains admit scheme A in their registry. `ProfileID` closes that gap.

Bullets aside, the two designs converge: a strict-PQ chain that wants to expose a 7932-style registry to other chains can declare exactly one entry (`TxAuthEnvelope`) and exactly one scheme (ML-DSA-65), and the externally-observable behaviour is equivalent. The Lux production canonical takes the simpler path: no in-chain registry, a single mandatory envelope.

3.6 Wire codec

`TxAuthEnvelope` serialises as a deterministic big-endian layout (`tx_envelope.go:178`):

Table 3: TxAuthEnvelope wire layout, fixed-width except `signature`.

Offset	Size	Field
0	2	`version` (uint16 BE)
2	1	`profile_id` (uint8)
3	4	`chain_id` (uint32 BE)
7	4	`network_id` (uint32 BE)
11	48	`account_id`
59	8	`nonce` (uint64 BE)
67	8	`expiry_height` (uint64 BE)
75	1	`wallet_scheme_id` (uint8)
76	1	`hash_suite_id` (uint8)
77	48	`fee_payer`
125	8	`gas_limit` (uint64 BE)
133	32	`max_fee`
165	32	`call_root`
197	32	`access_list_root`
229	32	`z_identity_root`
261	32	`account_state_root`
293	32	`public_key_ref`
325	4	`sig_length` (uint32 BE)
329	≤ 3309	`signature` (ML-DSA-65)

The 329-byte fixed header followed by a length-prefixed signature is the entire wire format. There is no nested encoding, no varints, no RLP ambiguity, no field-ordering choice. A naive decoder that consumes the bytes in order produces the canonical envelope; *any* ambiguity in decoding is a fatal decoder bug, not a protocol question.

4 PQPermit and the PQAuth precompile

4.1 What it replaces

EIP-2612 [23] is the canonical Ethereum permit primitive: a token holder signs an EIP-712 structured-data digest authorising a spender to move a value up to a deadline; the verifying contract calls `ecrecover` on `permit(owner, spender, value, deadline, v, r, s)` and grants the allowance. Every DeFi protocol with one-click flows sits on top of EIP-2612 or a near-equivalent.

EIP-712 [6] is structurally sound but uses ECDSA underneath. A Shor-capable adversary forges every EIP-2612 permit in flight. A strict-PQ chain needs a permit primitive that (1) signs under an ML-DSA-family scheme, (2) binds the verifying contract identity into the digest so a permit cannot be replayed across contracts, and (3) integrates with the `ChainSecurityProfile` so the contract-side verifier can refuse a permit signed under the wrong scheme.

`PQPermit` (`luxfi/consensus@v1.23.7`, `protocol/auth/permit.go:70`) is that primitive.

4.2 The 11 fields

Table 4: PQPermit field list. Source: `protocol/auth/permit.go:70`.

#	Field	Role
1	`Version`	permit version (uint16). $v = 1$ on strict-PQ.
2	`ProfileID`	profile under which signed.
3	`ChainID`	destination chain id.
4	`VerifyingContract`	48-byte contract address (PQ-native, not 20-byte).
5	`OwnerAccountID`	48-byte owner account identifier.
6	`Spender`	48-byte spender account identifier.
7	`Value`	32-byte permitted value (token-defined units).
8	`Nonce`	uint64 per-owner permit nonce.
9	`Deadline`	uint64 chain height after which permit invalid.
10	`AuthSchemeID`	contract auth scheme (matches contract's profile).
11	`HashSuiteID`	hash family (matches `profile.HashSuiteID`).

The 11 fields are the contract-permit equivalents of the 17 transaction fields, minus the consensus-side identifiers (`NetworkID`, `ZIdentityRoot`, `AccountStateRoot`, `CallRoot`, `AccessListRoot`, `FeePayer`, `GasLimit`, `MaxFee`, `ExpiryHeight`, `PublicKeyRef`) and plus the permit-side identifiers (`VerifyingContract`, `OwnerAccountID`, `Spender`, `Value`, `Deadline`). `Nonce` appears in both because both need per-account replay protection.

4.3 Digest

Listing 4: `Digest` for `PQPermit`. Source: `protocol/auth/permit.go:95`.

```
func (p *PQPermit) Digest() [48]byte {
    parts := [][]byte{
        []byte(pqPermitProtocolTag),     // "Lux/PQPermit/v1"
        u16BE(p.Version),
```

```
        {byte(p.ProfileID)},
        u32BE(p.ChainID),
        p.VerifyingContract[:],
        p.OwnerAccountID[:],
        p.Spender[:],
        p.Value[:],
        u64BE(p.Nonce),
        u64BE(p.Deadline),
        {byte(p.AuthSchemeID)},
        {byte(p.HashSuiteID)},
    }
    return tupleHash48(parts, pqPermitDigestCustomization)
}
```

Customization string: `LUX_PQ_PERMIT_V1`. Distinct from `LUX_TX_AUTH_V1` so a transaction-envelope signature cannot replay as a permit (or vice-versa), even if the data bytes happen to align.

4.4 ContractAuthProfile

A contract that consumes `PQPermit` declares its admitted schemes via a `ContractAuthProfile` (`protocol/auth/contract_profile.go`). The profile is a small struct carrying:

- `ContractAuthIDs`: the list of `AuthSchemeID` values the contract will accept (e.g., `{ML-DSA-65}`, optionally `{ML-DSA-65, ML-DSA-87}`).
- `HashSuiteID`: the hash family the contract accepts (must match `profile.HashSuiteID` on strict-PQ).
- `MinSecurityLevel`: a one-byte enum lower bound; refuses permits signed under a scheme below this floor.
- `AllowedPermitTypes`: a small bitmap of permit-type flags (e.g., transfer, approve, withdraw).

The verifier function `VerifyPQPermit` (`permit.go:VerifyPQPermit`) runs the same fail-closed sequence as the transaction verifier (Section 3): structural, profile/id match, profile/suite match, contract-profile match, deadline, nonce, signature.

4.5 The PQAuth precompile at 0x012202

EVM contracts call `PQAuth.verifyPermit(...)` as a Solidity library that internally dispatches to the `PQAuth` precompile at address `0x012202`. The precompile is the EVM-side entry point to the Go `VerifyPQPermit` (`luxfi/contracts@cb13a04`, `contracts/precompiles/PQAuth.sol`, `IPQVerify.sol`).

The split is deliberate:

- Pure-Solidity ML-DSA verification [45], even with EIP-8051-style expanded-pubkey form, costs *8.1M gas* on a single permit. That is unaffordable on a hot path.
- Native ML-DSA verification at the precompile, with the expanded-pubkey form, costs $\approx$ 130k *gas* per call (Section 12). That is 62$\times$ cheaper.
- The precompile bypasses Solidity opcode metering entirely for the lattice math; only the call dispatch and the result decoding count against the calling contract's gas budget.

The address `0x012202` is allocated in the `0x012200` block, which HIP-0084 §3 designates as the auth-precompile block. Adjacent precompiles (`0x012201`: signature batch verify; `0x012203`: session pubkey lookup) round out the auth surface.

4.6 IPQVerify interface

The Solidity interface is a single function:

Listing 5: IPQVerify.sol — contract-side entry point. Source: `luxfi/contracts@cb13a04`, `contracts/precompiles/IPQVerify.sol`.

```
interface IPQVerify {
    function verifyPermit(
        bytes calldata permit,        // wire-encoded PQPermit
        bytes calldata signature,     // ML-DSA-65 signature
        bytes32 expectedAccountId,    // OwnerAccountID anchor
        bytes32 expectedProfileHash   // ChainSecurityProfile digest
    ) external view returns (bool valid, uint256 reasonCode);
}
```

The contract is responsible for two things the precompile cannot do on its own: (1) supplying the `expectedAccountId` that the contract was constructed against, and (2) checking that the returned `reasonCode` is zero before granting the allowance. If either step is skipped, the contract has a logic bug; the precompile cannot defend against that. The PQAuth.sol library wraps the call in a fail-closed helper that enforces both.

4.7 Why expanded-pubkey is mandatory

EIP-8051 [15] proposed an expanded-pubkey form for ML-DSA verification on the EVM: the contract carries the full A-matrix expansion rather than the seed, in exchange for skipping the SHAKE expansion at verification time. The pubkey is much larger (60.6 KB for ML-DSA-65 expanded vs. 1.952 KB for the seeded form) but verification is cheaper by a constant factor.

Strict-PQ *requires* expanded-pubkey for the precompile path. The seeded form has variable verification cost (the SHAKE expansion runs every call); the expanded form has constant verification cost (just the lattice math). For a precompile metered in gas, predictable cost is more important than pubkey size. The contract's per-permit storage cost is one `bytes32` hash — a reference to the expanded pubkey held in a side-channel pubkey store. Lookups against the pubkey store are batched and amortised by the Z-Chain identity rollup (Section 5); the per-permit overhead is two storage reads.

4.8 Comparison to EIP-2612 economics

Table 5: Permit primitive comparison. Numbers from Section 12 bench harness; ECDSA cost is the canonical EVM `ecrecover` cost.

Primitive	Sig size	PQ?	Gas / verify	Replay protection
EIP-2612 + EIP-712 (ECDSA)	65 B	no	3,000	domain + nonce + deadline
ETHDILITHIUM Solidity	3309 B	yes	8,100,000	domain + nonce + deadline
`PQPermit` + `PQAuth.sol`	3309 B	yes	≈ 130,000	11 fields TupleHash256

The strict-PQ permit is 43× more expensive in verification gas than the classical ECDSA permit, but 62× cheaper than the naive pure-Solidity PQ alternative. The signature is 51× larger than ECDSA in bytes. Section 12 carries the full microbenchmark; the short answer is that the cost is acceptable on every contract that is willing to pay 130k gas for a one-click flow, which is approximately every DeFi user-facing contract.

5 Z-Chain: the identity rollup

5.1 Why a rollup

Sections 3 and 4 both leave one question open: *where does the verifier get the ML-DSA public key*? `TxAuthEnvelope.PublicKeyRef` is a 32-byte hash and `PQPermit.OwnerAccountID` is a

48-byte identifier; neither is the 1,952-byte (or 60.6-KB expanded) pubkey the lattice verifier needs.

The naive answer — carry the pubkey on the wire with every transaction or permit — is unaffordable. ML-DSA-65 pubkeys are 1,952 B; an expanded pubkey is 60.6 KB. A chain with 10^8 daily transactions at 1,952 B per pubkey is moving 186 GB of pubkey duplication *per day.*

The right answer is an identity rollup: pubkeys are registered once, revoked or rotated as needed, and committed via a Merkle root that `TxAuthEnvelope.ZIdentityRoot` binds. The verifier looks up the pubkey by `PublicKeyRef` against the rollup state at the bound root, with a small inclusion proof.

Z-Chain (`luxfi/consensus@v1.23.7`, `protocol/zchain/registry/`) is that rollup. HIP-0078 §4 defines its shape.

5.2 PQKeyRecord

The leaf of the identity rollup is `PQKeyRecord` (`protocol/zchain/registry/key_record.go`):

Listing 6: PQKeyRecord leaf. Source: `protocol/zchain/registry/key_record.go`.

```
type PQKeyRecord struct {
    AccountID       [48]byte     // owner identifier
    SchemeID        SigSchemeID  // ML-DSA-65 etc.
    CompactPubkey   []byte       // seed form (1952 B for ML-DSA-65)
    ExpandedRef     [32]byte     // SHA3-256 of the expanded-pubkey form
    Status          KeyStatus    // Active / Revoked / Rotated
    EpochRegistered uint64
    EpochExpiry     uint64       // 0 = no expiry
}
```

`Status` is one of `KeyStatusActive`, `KeyStatusRevoked`, `KeyStatusRotated`. The verifier accepts only `KeyStatusActive` records. `ExpandedRef` is the link into the side-channel pubkey store (Section 4); the precompile dereferences this when it needs the expanded form.

The leaf hash `PQKeyRecord.Hash()` is TupleHash256 over the leaf fields under customization `LUX_PQ_KEY_RECORD_V1`. Two leaves differing in any field have different hashes; therefore the Merkle tree detects any modification.

5.3 The five operations

The Z-Chain accepts five operation types (`protocol/zchain/registry/ops.go`). Each carries a TupleHash256 signing digest under an op-specific customization, plus a FIPS 204 §5.2 `ctx` of `LUX_ZCHAIN_REGISTRY_V1` (`ops.go:43`) on every ML-DSA signature. The op-specific customization plus the registry-wide `ctx` are independent layers of domain separation; replaying a signature from one op as another fails on both.

Table 6: Z-Chain registry operations. Source: `protocol/zchain/registry/ops.go`.

Op	Customization	Action
`OpRegisterKey`	`LUX_OP_REGISTER_KEY_V1`	Insert fresh key; proof of possession self-sig.
`OpRotateKey`	`LUX_OP_ROTATE_KEY_V1`	Replace one key with another; sig under outgoing key.
`OpRevokeKey`	`LUX_OP_REVOKE_KEY_V1`	Mark a key revoked; sig under the revoked key (plus profile-pinned reason).
`OpAuthorizeSession`	`LUX_OP_AUTHORIZE_SESSION_V1`	Bind an ephemeral session pubkey to a long-term account.
`OpCommitTxAuthBatch`	`LUX_OP_COMMIT_TX_AUTH_BATCH_V1`	Commit a batch of `TxAuthEnvelope` hashes for later inclusion proofs.
`OpCommitPermitBatch`	`LUX_OP_COMMIT_PERMIT_BATCH_V1`	Commit a batch of `PQPermit` hashes for later inclusion proofs.

Strictly six operations, not five, once `OpCommitPermitBatch` is counted alongside `OpCommitTxAuthBatch`. The HIP-0078 informal "five-op" framing groups the two commit ops because they share their verifier; the canonical six are documented in `ops.go`. We carry the informal framing in the paper title but use the precise count here.

5.3.1 OpRegisterKey

The proof of possession is the canonical: a registrant signs their own `AccountID` transcript under the key they're registering. Without this, an attacker could re-register a victim's pubkey as their own and intercept the victim's future transactions. The signing digest (`ops.go:66`) binds:

$$\text{digest} = \texttt{TupleHash256}\begin{pmatrix}\texttt{"Lux/OpRegisterKey/v1"},\\ \texttt{NewRecord.AccountID},\\ \texttt{NewRecord.Hash()}\end{pmatrix} \quad \text{cust} = \text{LUX\_OP\_REGISTER\_KEY\_V1}.$$

The customization plus the record hash plus the AccountID bind every byte that defines "this key, for this account, under this op".

5.3.2 OpRotateKey

`OpRotateKey` binds the outgoing record's hash, the incoming record's hash, and the AccountID. The signature is under the *outgoing* key: the existing key authorises its own succession. The verifier checks the chain (`outgoing.AccountID == incoming.AccountID`, `outgoing.SchemeID == incoming.SchemeID`, `outgoing.Status == Active`, `incoming.Status == Active`), then runs ML-DSA verify against the outgoing key. Atomically, the outgoing record transitions to `KeyStatusRotated`, and the incoming record becomes the active key for the AccountID.

5.3.3 OpRevokeKey

`OpRevokeKey` carries a profile-pinned `Reason` byte (compromise, lost, lifecycle, governance). The signature is under the revoked key. After acceptance, the record's `Status` is `KeyStatusRevoked` and the `RevocationRoot` aggregator (below) includes the leaf.

5.3.4 OpAuthorizeSession

A session key is an ephemeral pubkey with a TTL and a capability scope. The holder of the long-term identity key signs an authorisation binding the ephemeral pubkey hash, the expiry epoch, and the capability bitmap. The session key can sign transactions whose `TxAuthEnvelope.WalletSchemeID` matches the session scheme; the verifier looks up the session key against `SessionKeyRoot` and confirms the binding. Session keys do not appear in `IdentityRoot`; they appear in `SessionKeyRoot` only. The two roots are independent to support cheap session-key rotation without changing `IdentityRoot`.

5.3.5 OpCommitTxAuthBatch and OpCommitPermitBatch

These two ops commit Merkle roots over batches of accepted `TxAuthEnvelope` or `PQPermit` hashes, so the execution-side verifier on later chains (e.g., the EVM verifying a permit inside a transaction) can supply a Merkle inclusion proof rather than the full envelope. The batch root is signed by the consensus committee under the `FinalitySchemeID`; the committee is the same one running the Pulsar finality lane (Section 6).

5.4 ZRegistryRoots

The seven-root aggregator (`registry/roots.go:40`):

Listing 7: ZRegistryRoots aggregator. Source: `protocol/zchain/registry/roots.go:40`.

```
type ZRegistryRoots struct {
    IdentityRoot       [48]byte   // active PQKeyRecord set
    AccountKeyRoot     [48]byte   // same set indexed by AccountID
    RevocationRoot     [48]byte   // revoked / rotated AccountIDs
    SessionKeyRoot     [48]byte   // authorised session keys
    ContractPolicyRoot [48]byte   // per-contract policy hashes
    TxAuthRoot         [48]byte   // OpCommitTxAuthBatch roots
    PermitAuthRoot     [48]byte   // OpCommitPermitBatch roots
}

func (r *ZRegistryRoots) Hash() [48]byte {
    parts := [][]byte{
        []byte("Lux/ZRegistryRoots/v1"),
        r.IdentityRoot[:],  r.AccountKeyRoot[:],
        r.RevocationRoot[:], r.SessionKeyRoot[:],
        r.ContractPolicyRoot[:], r.TxAuthRoot[:],
        r.PermitAuthRoot[:],
    }
    return tupleHash48(parts, "LUX_ZCHAIN_REGISTRY_ROOTS_V1")
}
```

The aggregator hash rides as `EpochCommitment.zchain_state_root` on the Q-Chain. Q-Chain does not look inside; it consumes the 48-byte digest and binds it into `ComputeRoundDigest` (Section 6). Adding a root bumps the customization `V1` → `V2` and is a hard fork — the wire layout and the bound set are both pinned by the tag.

The seven roots are decomposed because the verifier hot path consumes only a subset:

- `IdentityRoot` is consumed by `TxAuthEnvelope` verification (Step 6 in Section 3).
- `AccountKeyRoot` is the AccountID-indexed lookup tree for execution-side calls.
- `RevocationRoot` is consumed when checking that an AccountID has no live revocation entries.
- `SessionKeyRoot` is consumed when validating a session-key-signed envelope.
- `ContractPolicyRoot` is consumed by `ContractAuthProfile` lookups.
- `TxAuthRoot` and `PermitAuthRoot` support inclusion-proof verification of historical envelopes.

5.5 VerifyAuthPassed: the hot-path inclusion verifier

`VerifyAuthPassed` (`registry/auth_proof.go:118`) is the verifier the EVM (and S-VM / X-VM) calls when it needs to confirm that a `TxAuthEnvelope` or `PQPermit` hash was committed by the Z-Chain. The function signature is roughly:

Listing 8: `VerifyAuthPassed` signature. Source: `protocol/zchain/registry/auth_proof.go:118`.

```
func VerifyAuthPassed(
    profile  *config.ChainSecurityProfile,
    root     [48]byte,           // ZRegistryRoots.TxAuthRoot or PermitAuthRoot
    leaf     [48]byte,           // hash of the envelope / permit
    proof    MerkleProof,
) (bool, error)
```

The verifier:

1. Confirms the proof's `HashSuiteID` matches `profile.HashSuiteID`.
2. Walks the proof, reconstructing the root via the leaf cust `LUX_MERKLE_LEAF_V1` and the node cust `LUX_MERKLE_NODE_V1`.
3. Returns `(true, nil)` iff the reconstructed root equals the bound root.

The two customizations are critical: a leaf hash cannot be confused with an internal-node hash, so a proof cannot be smuggled by re-interpreting a leaf as an internal node (the classical "second preimage on Merkle trees" attack). cSHAKE256 with distinct customizations prevents the collision structurally.

5.6 Why this is a rollup, not a state tree

The seven roots aggregate *commitments*, not state. The full identity state lives in the Z-Chain's own state tree. The seven roots are EpochCommitment-level digests that the Q-Chain finalises. This is the same pattern an optimistic / ZK rollup follows on Ethereum: the rollup chain runs its own VM and produces a commitment that the L1 chain accepts.

The difference is that Z-Chain is co-validated by the same committee that validates the C-Chain, P-Chain, and X-Chain — there is no separate rollup operator. The finality lane (Section 6) signs every Z-Chain commitment with Pulsar-65, the same lane that finalises the other chains. The result is a single PQ identity layer shared by every execution chain on Lux.

6 Q-Chain finality: the 23-axis binding

6.1 What Q-Chain binds

Q-Chain is the chain that produces finality certificates over every other chain in the Lux ecosystem. HIP-0079 defines Q-Chain as the "post-quantum finality plane": it consumes EpochCommitments from every execution chain (C-Chain, P-Chain, X-Chain, Z-Chain), folds them into a canonical QBlock, and produces a Pulsar finality certificate over the QBlock digest.

The QBlock digest is the single 48-byte commitment that says "this is the state of the Lux Network at epoch N under LUX_STRICT_E2E_PQ." Everything else — the bridge admission, the cross-chain Warp envelope, the Z-Chain commitment binding into a `TxAuthEnvelope` verification — ultimately points back to this digest.

6.2 The 23 axes

`ComputeRoundDigest` (`luxfi/consensus@12097034`, `protocol/qchain/round_digest.go`) binds 23 axes into the QBlock hash. The axes are the security-relevant inputs to consensus at this round:

Table 7: ComputeRoundDigest axes. The 23 axes are the inputs to consensus at a single round. Adding an axis bumps the customization `LUX_QBLOCK_V1` → `V2` and is a hard fork.

#	Axis	Role
1	`ProfileHash`	ChainSecurityProfile digest.
2	`Version`	QBlock version.
3	`NetworkID`	destination network.
4	`Epoch`	monotonic epoch counter.
5	`Height`	Q-Chain height.
6	`PrevBlockHash`	parent QBlock digest.
7	`CChainCommitment`	C-Chain EpochCommitment digest.
8	`PChainCommitment`	P-Chain EpochCommitment digest.
9	`XChainCommitment`	X-Chain EpochCommitment digest.
10	`ZChainCommitment`	Z-Chain EpochCommitment digest (`ZRegistryRoots.Hash()`).
11	`ValidatorSetRoot`	active validator set Merkle root.
12	`ValidatorStakeRoot`	per-validator stake commitment.
13	`BridgeAdmissionRoot`	bridge admission state root.
14	`TxAuthBatchRoot`	accepted `TxAuthEnvelope` root for epoch.
15	`PermitBatchRoot`	accepted `PQPermit` root for epoch.
16	`ZIdentityRoot`	shortcut into `ZRegistryRoots.IdentityRoot`.
17	`P3qProofRoot`	p3q STARK proof aggregator root for this epoch.
18	`RandomnessSeed`	DRBG seed for this epoch.
19	`PrevFinalityCert`	previous epoch's Pulsar cert digest.
20	`Timestamp`	uint64 epoch timestamp (UTC seconds).
21	`LineageRoot`	lineage chain digest (HIP-0079 §5).
22	`GovernanceRoot`	active governance state root.
23	`ExtraHash`	48-byte HIP-future-reserved digest.

Every axis except `ExtraHash` is consumed by some downstream verifier. `ExtraHash` is reserved for future HIP additions: future proposals will bind new axes into the `ExtraHash` preimage so the QBlock layout itself stays stable across the migration window.

6.3 ComputeRoundDigest

The construction:

Listing 9: `ComputeRoundDigest` for QBlock. Source: `luxfi/consensus@12097034`, `protocol/qchain/round_digest.go`.

```
func (q *QBlock) ComputeRoundDigest() [48]byte {
    parts := [][]byte{
        []byte("Lux/QBlock/v1"),
        q.ProfileHash[:],
        u16BE(q.Version),
        u32BE(q.NetworkID),
        u64BE(q.Epoch),
        u64BE(q.Height),
        q.PrevBlockHash[:],
        q.CChainCommitment[:], q.PChainCommitment[:],
        q.XChainCommitment[:], q.ZChainCommitment[:],
        q.ValidatorSetRoot[:], q.ValidatorStakeRoot[:],
        q.BridgeAdmissionRoot[:],
        q.TxAuthBatchRoot[:], q.PermitBatchRoot[:],
        q.ZIdentityRoot[:], q.P3qProofRoot[:],
        q.RandomnessSeed[:],
```

```
        q.PrevFinalityCert[:],
        u64BE(q.Timestamp),
        q.LineageRoot[:],
        q.GovernanceRoot[:],
        q.ExtraHash[:],
    }
    return tupleHash48(parts, "LUX_QBLOCK_V1")
}
```

The Pulsar finality lane (Section 7) signs over this digest. The signature, together with the QBlock body, is the finality certificate.

6.4 Canonical QBlock layout

The QBlock body is the wire-serialised form of the struct above plus the finality certificate. The wire layout is deterministic big-endian, fixed-width except for the certificate trailer:

Table 8: QBlock wire layout. The fixed header is 1,080 bytes; the certificate trailer is the Pulsar aggregate plus committee bitmap, sized per the active committee (typically ≈ 33 KB).

Offset	Size	Field
0	48	`profile_hash`
48	2	`version`
50	4	`network_id`
54	8	`epoch`
62	8	`height`
70	48	`prev_block_hash`
118	48	`cchain_commitment`
⋮	⋮	⋮ (per Table 7)
1,072	8	`timestamp`
1,080	varies	`finality_cert` (Pulsar-65 aggregate + bitmap)

6.5 Finality cert binding to Pulsar

The Pulsar finality lane (Section 7) operates as a two-round threshold signature scheme over the 43-of-64 active committee. Round 1 commitments hash the QBlock digest as their session id. Round 2 opens partial signatures that aggregate linearly into a single 33-KB certificate. The verifier:

1. Recomputes the QBlock digest from the body.
2. Looks up the active committee against `ValidatorSetRoot`.
3. Verifies the Pulsar aggregate signature over the digest under the committee's group public key (which is itself anchored in `ZRegistryRoots.IdentityRoot` from the Z-Chain).
4. Confirms the committee bitmap has at least 43 set bits.

Step 2 binds the certificate's committee to the consensus state *at this epoch*. Step 3 verifies the lattice math. Step 4 enforces the threshold. The certificate is binding under MLWE.

6.6 Why 23 axes specifically

The 23-axis count is the minimum closure of the dependency graph between the Lux chains. Each axis is the only commitment from *one* subsystem the verifier needs to confirm that subsystem's state matches what the QBlock claims. Dropping any axis would force a downstream verifier to consult the subsystem's own state instead of the QBlock, breaking the "QBlock is the single binding" invariant.

Adding axes is forward-compatible via the `ExtraHash` reservation; the next HIP that adds axes binds them into the `ExtraHash` preimage and ships an `ExtraHashV2` struct with the customization `LUX_QBLOCK_EXTRA_V2`. The QBlock layout itself does not change until the migration window closes and a hard-fork to `LUX_QBLOCK_V2` absorbs the new fields directly.

7 Pulsar: Module-LWE threshold finality

7.1 Why Pulsar and not BLS

The finality lane in classical Lux Quasar 3.0 [24] was BLS12-381 [8]: a 48-byte aggregate that closes a block in sub-second. BLS is the right shape (linear aggregation, small certificate, short verification) but the wrong primitive: it falls to Shor.

Pulsar is the PQ replacement. It is a two-round threshold signature over the cyclotomic ring $R_q = \mathbb{Z}_q[X]/(X^\varphi + 1)$ with $\varphi = 256$ and a 48-bit NTT-friendly prime q. The structure mirrors FROST [22]: Round 1 is offline, Round 2 binds the message, partial signatures aggregate linearly. The lattice instantiation is closest to the two-round threshold ML-DSA family [9, 2] but tracks the hint-bound discipline of [7].

The reference implementation is `luxfi/pulsar`. The production canonical exposes `keygen`, `sign`, `verify`, `threshold`, `dkg`, `reshare`, and `abort` operations.

7.2 Three security levels

Pulsar ships three security levels:

Table 9: Pulsar security levels. Parameters from `luxfi/pulsar/parameters.go`.

Variant	Module rank	Aggregate size	Use case
Pulsar-44	4	≈ 22 KB	devnet / fuzzing only
Pulsar-65	6	≈ 33 KB	production finality lane
Pulsar-87	8	≈ 44 KB	high-value treasury / governance

The strict-PQ profile pins `FinalitySchemeID = Pulsar-65` and `HighValueSchemeID = Pulsar-65 or Pulsar-87`. Pulsar-44 is *forbidden* on locked profiles — it exists for devnet stress testing where the 22 KB cert and faster signing cost matter more than the 128-bit security floor.

7.3 Two-round protocol

The protocol structure (Algorithm 1) is near-identical to the original Pulsar family [25]; the difference is the choice of NTT modulus and the hint-bound discipline. We sketch the construction; the reference impl carries the byte-exact specification.

The verifier (`luxfi/pulsar/verify.go`) reconstructs $\mathbf{A}z - \tilde{\mathbf{b}}c$ in the rounded-coefficient hint domain R_ν, applies the per-coefficient Δ correction, and accepts iff the L_2 bound holds. The bitmap is the indicator vector for which parties contributed — at least 43 of 64 must be set.

The threshold $t = 43$ is chosen against $n = 64$ to give:

- BFT margin: $t > 2n/3$ closes the standard Byzantine quorum.
- Identifiable abort: $n - t = 21$ allows up to 21 non-contributing parties to be identified without aborting the round.
- Lattice slack: the hint discipline tolerates t-of-n aggregation without rejection-sampling rerolls in the $\eta_\varepsilon = 2.65$ slack regime.

Algorithm 1 Pulsar-65 two-round signing

Require: committee $\mathcal{C}$ of $n = 64$ parties; threshold $t = 43$
Require: message m (typically the QBlock digest, Sec. 6)
Require: group public key $(\mathbf{A}, \tilde{\mathbf{b}})$ from DKG
Require: each party holds Shamir share $\mathbf{s}_i$

Round 1 (offline, per party i):
  sample masking matrix $\hat{R}_i$ from Discrete-Gaussian σ_E
  broadcast $D_i = \mathbf{A} \cdot \hat{R}_i + \hat{E}_i$
  attach BLAKE3-keyed MAC $\mathrm{MAC}_{i \to j}$ for each peer j
Round 2 (online, per party i):
  aggregate $D = \sum_j D_j$
  derive challenge vector $u = \textsc{GaussianHash}(D, m, \texttt{ctx})$
  derive ternary challenge $c = \textsc{ChallengeHash}(u, m)$
  compute partial signature $z_i = \mathbf{s}_i \cdot c + \hat{R}_i$
  broadcast z_i
Finalise:
  aggregate $z = \sum_i z_i$ (linear, no coordinator)
  produce hint Δ in R_ν
  output certificate $(c, z, \Delta, \text{bitmap})$

7.4 Identifiable abort

A party that misbehaves in Round 1 (publishes a malformed D_i, fails its MAC) or Round 2 (publishes a malformed z_i) is identified by the protocol without halting the round. The reference impl exposes the `abort` interface (`luxfi/pulsar/abort.go`) that returns the identifier of the misbehaving party. The consensus engine then ejects the party from the active committee and slashes their stake.

This is critical for liveness: a BLS threshold signature that fails to aggregate has no identifiable culprit and the whole round must be re-run. Pulsar rounds make progress against $n - t$ misbehaving parties without re-runs, which is the practical difference between sub-second finality and multi-second finality under partial failure.

7.5 No BLS fallback

The strict-PQ profile sets `ForbidFallbacks = true`. This bit specifically refuses the "fall back to BLS if Pulsar times out" design the early Quasar drafts considered. The reasoning: a Shor-capable adversary could trigger Pulsar failure on purpose to force the fallback path, then forge under BLS. Refusing the fallback closes the attack at the cost of slower finality during a Pulsar misconfiguration.

The replacement for the BLS fallback is the identifiable-abort path above: slow-but-correct under failure, not fast-but-classical.

7.6 DKG and reshare

Validator-set rotation requires resharing the Pulsar secret to a new committee. The reference impl exposes `dkg` (`luxfi/pulsar/dkg.go`) and `reshare` (`reshare.go`). The DKG is a Pedersen-style protocol over R_q with hiding / binding proofs, closely following the construction in [21, 17]. Reshare uses the LSS framework [41] so the group public key $(\mathbf{A}, \tilde{\mathbf{b}})$ is preserved across the committee transition.

The activation certificate (HIP-0078 §6) gates each committee transition on a Pulsar signature verifying under the unchanged group public key. Activation proves the new committee *can*

sign before the old committee loses signing capability; complaint and slashing-evidence pipelines are independent.

7.7 Reference impl status

`luxfi/pulsar` ships:

- `keygen`: long-term signing key generation.
- `sign`: single-party signing under a non-threshold key (for testing).
- `verify`: standalone verifier.
- `threshold`: the two-round protocol of Algorithm 1.
- `dkg`: distributed key generation.
- `reshare`: secret-share resharing across committee transitions.
- `abort`: identifiable-abort detection.

Every operation has KAT (known-answer tests) vectors in the `testdata/` directory. The KATs are generated by the reference impl itself and cross-checked against the C++ port (`luxfi/pulsar-cpp`, in development) and against the GPU kernels (`luxfi/p3q-gpu` for the NTT subroutines).

8 p3q: STARK / FRI over SHA3

8.1 Why not KZG, BN254, or Groth16

The classical SNARK stack on Ethereum is built on KZG polynomial commitments [20] over BN254 [3], with Groth16 [19] as the PLONK / R1CS proof wrapper. Every component of that stack is classical:

- BN254 is a pairing-friendly elliptic curve; Shor breaks the discrete log; the security claim is gone.
- KZG requires a structured reference string (SRS). Compromising the SRS (via toxic waste from the trusted setup) forges proofs.
- Groth16 requires a per-circuit SRS, multiplying the trusted-setup risk per deployed circuit.

A strict-PQ chain refuses the entire stack. The replacement is a STARK / FRI proof system over SHA3 Merkle commitments:

- **Transparent.** No SRS, no trusted setup.
- **Hash-based.** Soundness reduces to collision-resistance of cSHAKE256 / SHA3-384, which is conjectured PQ-secure.
- **FRI-based.** The polynomial-commitment trapdoor is replaced by the Fast Reed-Solomon Interactive Oracle Proof of Proximity, which is collision-resistant under the same hash assumption.

The Lux reference is p3q (`luxfi/p3q@v0.0.1`), ten crates shipped on crates.io.

8.2 The ten crates

Table 10: p3q v0.0.1 crate inventory. All crates published on crates.io; canonical at https://github.com/luxfi/p3q.

Crate	Role
`p3q-core`	shared types, traits, error model
`p3q-field`	PQ-friendly finite-field arithmetic (BabyBear / KoalaBear / Goldilocks variants)
`p3q-sha3`	SHA3-384 / cSHAKE256 wrappers, KECCAK-permutation primitives
`p3q-merkle`	SHA3 Merkle trees with leaf/node cust separation
`p3q-challenger`	Fiat-Shamir challenger over cSHAKE256
`p3q-fri`	Fast Reed-Solomon IOP of Proximity
`p3q-stark`	STARK proof construction & verification
`p3q-recursion`	recursive STARK composition
`p3q-verifier`	no-std verifier crate for embedded / EVM / WASM targets
`p3q-zchain`	Z-Chain rollup state-transition prover

The crate split is the Rust idiom (small crates, clear dependencies). The `p3q-verifier` crate is `no_std` so it can be embedded in WASM, in the EVM precompile, and in mobile-class verifiers. The `p3q-zchain` crate is the Z-Chain-specific state-transition prover; other chains' provers (Q-Chain finality validation, C-Chain state proof) will live in their own `p3q-*chain` crates as they ship.

8.3 Profile binding

`ChainSecurityProfile.AllowedProofBackends` pins the admitted backends. Under strict-PQ the list is {`p3q`, `Plonky3-SHA3`}:

- `p3q` is the production-default canonical.
- `Plonky3-SHA3` is the SHA3-Merkle variant of Plonky3 [38], listed as a permitted alternative for circuits that are easier to express in Plonky3's AIR than in p3q's. The `-SHA3` suffix is the SHA3-Merkle variant exclusively; the classical Plonky3-Poseidon variant is *not* permitted because the Poseidon hash family is not part of FIPS 202.

`AllowedProofFormats` pins `p3q-v0.1`, the v0.1 wire format. Future format versions bump the format ID.

The profile floors `MinSoundnessBits` = 128 (FRI-soundness $\geq$ 128 bits) and `MinHashOutputBits` = 384 (SHA3-384). The `ForbidPairings`, `ForbidKZG`, `ForbidTrustedSetup`, `ForbidClassicalSNARKs` bits all hold; any backend that wraps a STARK in a Groth16 verifier to cheapen EVM verification is refused.

8.4 The expanded-pubkey wire form

The verifier consumes proofs in EIP-8051-style *expanded-pubkey* form. EIP-8051 [15] was originally proposed for ML-DSA verification: the contract carries the full A-matrix expansion rather than the seed, in exchange for skipping the SHAKE expansion at verification time. p3q generalises this to proof verification: the proof carries pre-expanded commitment data so the verifier never re-expands a seed at proof check time.

Concretely, an expanded p3q proof carries:

- All Merkle commitment trees in pre-built form.
- All FRI query positions resolved into leaf-path bundles.
- All Fiat-Shamir challenges resolved against the bound challenger state.

The verifier walks the pre-built structure with no PRG / SHAKE expansion in the hot loop. The result is that a STARK verifier in `no_std` is predictable in cost (no allocation, no SHAKE expansion), which makes it suitable for the EVM precompile path.

The cost is proof size: an expanded proof is $\approx 5\times$ larger than a seeded proof. The trade-off is the same as the ML-DSA expanded-pubkey trade: predictability over compactness.

8.5 Proof verification cost

Table 11: p3q proof verification cost (single STARK). Benchmark hardware: Apple M2 Pro, 12 cores, 32 GB. Numbers from `luxfi/p3q@v0.0.1/bench/` harness. Comparison to Plonky3 / Halo2 / Groth16 for context; Groth16 is forbidden by profile.

Backend	Proof size	Verify time	Verify gas (EVM, expanded)
`p3q v0.0.1`	$\approx$ 290 KB	7 ms	$\approx$ 750,000
`Plonky3-SHA3`	$\approx$ 260 KB	9 ms	$\approx$ 820,000
`Halo2`	$\approx$ 60 KB	18 ms	$\approx$ 1,500,000
`Groth16`	256 B	1.5 ms	$\approx$ 250,000

p3q is $3\times$ more expensive than Groth16 in EVM gas and is much larger on the wire, but Groth16 is *forbidden*. Among the permitted choices, p3q is the cheapest verifier and the smallest proof. The cost is acceptable for the rollup commit path (one verify per epoch) and for the high-value contract path; it is not acceptable for the per-transaction path, which uses `TxAuthEnvelope`-direct signature verification.

8.6 Recursion

`p3q-recursion` supports recursive STARK composition: a verifier circuit that verifies a previous proof inside a new proof. The use cases are:

- Folding a sequence of Z-Chain epoch transitions into a single aggregate proof.
- Bridge-side verification of a Lux chain state proof inside a proof on the destination chain.
- Cross-chain Warp envelope verification (HIP-0080) where the source-chain finality certificate is verified inside a destination-chain proof.

Recursion in p3q does not require a new SRS (the construction is transparent). The recursion overhead is approximately $2\times$ the base proof verification cost.

9 P2P: ML-KEM + ML-DSA peer handshake

9.1 Why the P2P layer matters

A chain whose consensus, transaction authorisation, and contract permits are all post-quantum is still classically vulnerable if peers connect over a classical Diffie-Hellman handshake. A Shor-capable adversary on the wire breaks the handshake, recovers the session key, and replays / forges peer messages with the same confidentiality guarantees as the original. The peer authentication primitive is the gating point.

Classical Lux peers used ed25519-static-keys + X25519-ephemeral-keys for the Noise-XK handshake. Both are classical: Shor breaks them. Strict-PQ replaces both: ML-KEM for the key exchange, ML-DSA for the static identity. The reference impl is `luxfi/node@dc906d281b`.

9.2 Construction

The handshake is a Noise-XK variant where:

- The static identity key is ML-DSA-65 (matching the chain's `IdentitySchemeID`).
- The ephemeral key encapsulation is ML-KEM-768 by default, ML-KEM-1024 for high-value links (validator-to-validator, bridge-to-bridge).
- The session key derivation hashes through five cSHAKE customizations that domain-separate the handshake stages.

The five cSHAKE customizations are:

Table 12: P2P handshake cSHAKE customizations. Each stage of the handshake uses an independent customization to prevent cross-stage replay.

Customization	Stage
`LUX_P2P_HANDSHAKE_INIT_V1`	initial KEM ciphertext binding
`LUX_P2P_HANDSHAKE_RESP_V1`	responder KEM commitment binding
`LUX_P2P_HANDSHAKE_AUTH_V1`	static-key authentication binding
`LUX_P2P_HANDSHAKE_KEY_V1`	session key derivation
`LUX_P2P_HANDSHAKE_REKEY_V1`	rekey rotation derivation

Each cSHAKE call binds the customization string plus the relevant transcript bytes. The handshake transcript binds: protocol version, both parties' ML-DSA-65 identity-pubkey hashes, ML-KEM-768 / 1024 ciphertext, the responder's confirmation byte, and the negotiated session parameters.

9.3 Why ML-KEM-768 by default and ML-KEM-1024 for high-value links

ML-KEM-768 is NIST Level 3 (192-bit). ML-KEM-1024 is NIST Level 5 (256-bit). The cost ratio is:

Table 13: ML-KEM cost comparison. Numbers from FIPS 203 §8.2 reference impl on commodity x86-64.

Variant	Pubkey	Ciphertext	Keygen + Encaps + Decaps (μs)
ML-KEM-512	800 B	768 B	25
ML-KEM-768	1,184 B	1,088 B	35
ML-KEM-1024	1,568 B	1,568 B	55

For peer-to-peer links among $\sim 10^4$ peers handshaking $\sim 10^2$ times per peer per hour, the bandwidth difference between ML-KEM-768 and ML-KEM-1024 is ~ 2 MB / hour — negligible. For validator-to-validator links handshaking many times per minute and carrying high-value finality traffic, the 256-bit security floor is worth the cost. The strict-PQ profile pins per-link KEM choice based on the link's classification (validator, bridge, regular peer); the default is ML-KEM-768.

9.4 Handshake flow

The Noise-XK variant:

Two ML-KEM encapsulations form the session key. The first binds the responder's identity; the second binds the initiator's identity. ML-DSA-65 authenticates the initiator over the bound transcript. The responder is authenticated by Noise-XK's a-priori-pubkey property (the initiator confirms PK_R during step 5).

Algorithm 2 Strict-PQ peer handshake (Noise-XK over ML-KEM + ML-DSA-65)

Require: Initiator I with static ML-DSA-65 key $(\mathrm{SK}_I, \mathrm{PK}_I)$
Require: Responder R with static ML-DSA-65 key $(\mathrm{SK}_R, \mathrm{PK}_R)$
Require: Initiator knows PK_R a priori (Noise-XK property)
1: I samples ephemeral ML-KEM keypair $(\mathrm{KPK}_I, \mathrm{KSK}_I)$
2: I sends $\langle \mathrm{KPK}_I, \mathrm{TAG}_1 \rangle$ where $\mathrm{TAG}_1 = \mathrm{cSHAKE256}(\texttt{INIT}, \mathrm{KPK}_I)$
3: R encapsulates against KPK_I, deriving shared secret s_1
4: R sends $\langle \mathrm{CT}_R, \mathrm{AEAD}_{s_1}(\mathrm{PK}_R), \mathrm{TAG}_2 \rangle$ where $\mathrm{TAG}_2 = \mathrm{cSHAKE256}(\texttt{RESP}, \mathrm{CT}_R, \mathrm{ENC}(\mathrm{PK}_R))$
5: I decapsulates, recovers s_1, AEAD-decrypts PK'_R, confirms $\mathrm{PK}'_R = \mathrm{PK}_R$
6: I samples second ephemeral ML-KEM keypair, encapsulates against PK_R to derive s_2
7: I sends $\langle \mathrm{CT}_I, \mathrm{AEAD}_{s_2}(\mathrm{PK}_I, \mathrm{SIG}_I), \mathrm{TAG}_3 \rangle$ where $\mathrm{SIG}_I = \text{ML-DSA-65.}\mathrm{Sign}_{\mathrm{SK}_I}(\mathrm{cSHAKE256}(\texttt{AUTH}, \ldots))$
8: R verifies SIG_I, derives session key $K = \mathrm{cSHAKE256}(\texttt{KEY}, s_1, s_2)$
9: Both parties begin AEAD-protected data exchange under K
10: Rekey at message-count or time-bound thresholds using `REKEY`

9.5 Five customizations, not one

Why five customization strings? The naive design uses a single `LUX_P2P_V1` customization for every cSHAKE call. The five-cust design is the audit-driven minimum that closes:

- **Cross-stage replay** (between INIT, RESP, AUTH, KEY, REKEY) is closed by per-stage customizations.
- **Initiator-impersonating-responder** attacks (where an attacker sends an INIT-tagged message to a peer expecting RESP) are closed by distinct INIT and RESP customizations.
- **Rekey-as-fresh-session** attacks (where a rekeyed session is misinterpreted as a fresh session) are closed by the dedicated REKEY customization.

The five-cust design is the Noise-protocol-framework minimum applied verbatim to cSHAKE256.

9.6 Static identity in Z-Chain

Peer static ML-DSA-65 keys are registered in the Z-Chain identity rollup under `OpRegisterKey` alongside transaction-signing keys. The `AccountID` of a peer is the chain's canonical peer identifier (which HIP-0077 §3 names `TypedNodeID` — see Section 10). The handshake's static-pubkey verification walks back to the Z-Chain identity rollup; the peer is admitted iff its pubkey is on the active set at the current epoch.

This integrates the P2P identity primitive with the chain identity primitive: a peer that loses validator status (revoked, rotated) is automatically dropped from the peer set at the next epoch transition. The current-epoch `IdentityRoot` is the source of truth.

9.7 No classical fallback

The handshake refuses classical primitives. There is no Curve25519 fallback, no ed25519 fallback, no AES-GCM-with-classical-KEM mode. A peer that advertises only classical primitives is refused at the version negotiation step (the protocol version byte is checked before any KEM / DSA work, so the refusal is cheap).

Classical-compat peers (running the permissive profile) reach the strict-PQ chain only through a translation gateway — they cannot directly join the peer mesh.

10 C-Chain / P-Chain / X-Chain coverage

10.1 Three execution chains, one identity layer

The Lux network ships three execution chains and one identity chain:

- **C-Chain**: EVM-compatible smart-contract chain.
- **P-Chain**: platform / staking / governance chain.
- **X-Chain**: UTXO-style asset chain.
- **Z-Chain**: PQ identity rollup (Section 5).

Each execution chain has its own VM, its own state tree, and its own finality cadence, but they share a single identity layer (Z-Chain), a single finality lane (Q-Chain via Pulsar), and a single security profile. This section covers what changes on each chain under LUX_STRICT_E2E_PQ.

10.2 TypedNodeID and ValidatorSchemeID

Classical Avalanche-family node identifiers were `NodeID = SHA256(ECDSA pubkey)[:20]` — a 20-byte hash of the classical pubkey. The hash is fine but the pubkey it indexes is classical.

The strict-PQ replacement is `TypedNodeID` (`luxfi/ids@v1.2.10`, `node_id_typed.go`). A `TypedNodeID` carries:

- A 1-byte `ValidatorSchemeID` indicating which scheme the underlying pubkey belongs to.
- A 48-byte digest of the pubkey under `cSHAKE256(LUX_NODE_ID_V1)`.

The `ValidatorSchemeID` is a registry-style byte (HIP-0077 §2):

Table 14: Validator scheme registry under LUX_STRICT_E2E_PQ.

ID	Scheme
`0x00`	invalid (zero-value-invalid; refused at boot)
`0x40`	ML-DSA-65 (strict-PQ default)
`0x41`	ML-DSA-87 (high-value variant)
`0x42`	ML-DSA-44 (devnet-only; refused on strict profile)
`0x50`	Pulsar-65 (finality lane only)
`0x51`	Pulsar-87
`0x80`	SLH-DSA-SHAKE-256s (FIPS 205 fallback identity)

The `0x40` block is ML-DSA (FIPS 204), the `0x50` block is Pulsar, the `0x80` block is SLH-DSA (FIPS 205). The block structure is HIP-0077 §2's design: each high nibble denotes a primitive family, the low nibble the security level.

The strict profile pins `IdentitySchemeID = 0x40` (ML-DSA-65). `ValidatorSchemeID` on every `TypedNodeID` appearing in the validator set must match the profile.

10.3 SchemeGate at boot

`SchemeGate` (`luxfi/node@v1.26.10`, `snow/engine/scheme_gate.go`) is the boot-time gate that refuses to start a node whose ValidatorSchemeID doesn't match the profile. The gate runs after profile load and before validator-set construction:

1. Load profile, compute hash, confirm against genesis pin.
2. For each entry in the genesis validator set:
   a. Parse `TypedNodeID`.
   b. Confirm `ValidatorSchemeID == profile.IdentitySchemeID`.
   c. If not, abort with `SCHEME_GATE_MISMATCH` fatal.
3. Proceed to consensus.

The gate is the canonical answer to "what if the genesis is wrong": the node refuses to start, the operator sees a clear error, and no chain state is corrupted by a misconfigured boot.

10.4 C-Chain: EVM-compatible PQ

The C-Chain runs the EVM. The strict-PQ posture on the C-Chain has four moving parts:

- Every transaction is a `TxAuthEnvelope` (Section 3). The mempool admits no other transaction type under `RequireTypedTxAuth = true`.
- Smart contract permits use `PQPermit` via the `PQAuth` precompile (Section 4).
- The signature precompiles for ML-DSA-65, ML-KEM-768, and SLH-DSA are exposed at well-known addresses (HIP-0084).
- The `mldsafx` fork — the multi-EVM hard fork (HIP-0089) — closes legacy precompiles that depend on classical primitives (Groth16 verifier, KZG point evaluation).

The `mldsafx` fork is the C-Chain's strict-PQ inflection. Before the fork, the EVM admits both classical and PQ transactions through the `ClassicalCompatRegistry` (below); after the fork, the registry is empty and only `TxAuthEnvelope` transactions exist.

10.5 ClassicalCompatRegistry: the transition mechanism

A migration path is impossible without admitting classical transactions during the migration window. The `ClassicalCompatRegistry` (`luxfi/node`, `mempool/registry.go`) is the allowlist of classical transaction types the mempool admits while the network is on the permissive profile.

The registry is a small list:

- `LegacyRLP` (Ethereum tx-type 0x00, raw RLP) — *admitted only during the migration window.*
- `EIP2930` (tx-type 0x01, access list) — admitted.
- `EIP1559` (tx-type 0x02, dynamic fee) — admitted.
- `EIP4844` (tx-type 0x03, blob) — admitted on the blob-enabled testnets; not on mainnet C-Chain at this writing.
- `TypedTxAuth` (Lux-canonical tx-type 0x04, `TxAuthEnvelope`) — always admitted; the only type on strict-PQ.

EVM tx-type `0x05` is a placeholder reserved for a future `TxAuthEnvelopeV2` variant (HIP-0089-future scoping). The current strict-PQ profile accepts `0x04` only.

On the strict-PQ profile, `RequireTypedTxAuth = true` forces the registry to a single-entry set $\{0x04\}$. Any classical tx-type submitted to the mempool is rejected before signature verification with reason code `TX_TYPE_FORBIDDEN`.

10.6 P-Chain and X-Chain

The P-Chain (platform / staking) and X-Chain (UTXO assets) follow the same pattern as the C-Chain:

- All P-Chain operations (validator registration, delegation, reward claim, governance) are wrapped in `TxAuthEnvelope`-equivalents — the P-Chain VM consumes the same envelope shape with chain-specific `CallRoot` interpretations.
- All X-Chain UTXO spends sign under the asset's owner's ML-DSA-65 key via `TxAuthEnvelope`. The asset itself can declare its own `ContractAuthProfile` (Section 4) for high-value spends.
- The classical-compat registry is empty on both chains under strict-PQ.

The two chains do not have a fork name equivalent to `mldsafx` (the C-Chain's term); the equivalent transition on P-Chain is the `PChainPQActivation` epoch and on X-Chain is the `XChainPQEpoch`. Both are pinned in the genesis configuration and both gate the activation on the profile being LUX_STRICT_E2E_PQ.

10.7 S-VM mempool gating

Sandbox VMs (S-VMs) running on top of the C-Chain follow the same `TxAuthEnvelope`-only rule. The S-VM mempool admits a transaction iff:

1. The outer C-Chain transaction is a `TxAuthEnvelope`.
2. The inner S-VM call carries an S-VM-specific transcript that the S-VM verifies under the S-VM's own scheme (which inherits from the outer profile but may declare additional fields).

A failed S-VM admission is reported as part of the outer transaction receipt; the outer transaction's gas is consumed, but the S-VM state is not modified. This is the standard EVM revert semantics applied to S-VM.

10.8 Mempool gating summary

Table 15: Mempool admission under `RequireTypedTxAuth = true`.

Chain	Admitted type	Refused types
C-Chain	`TypedTxAuth` (0x04)	0x00, 0x01, 0x02, 0x03, 0x05+
P-Chain	`TypedTxAuth` (P-Chain variant)	all legacy P-Chain types
X-Chain	`TypedTxAuth` (X-Chain variant)	all legacy X-Chain types
S-VM	inherits outer C-Chain	all

This is the canonical statement: under strict-PQ, exactly one mempool entry class exists per chain. There is no admission of any other type.

11 Bridge: refused primitives and the PQ admission profile

11.1 Why bridges are different

The bridge surface is the only Lux surface where the counterparty is not a Lux subsystem. Every other transcript in the strict-PQ profile is signed by a key registered in the Z-Chain identity rollup; bridges transact with external chains, external custodians, and external admission keys.

A strict-PQ chain can choose to refuse classical bridges entirely (the maximally-paranoid posture: no classical signer is ever admitted), or it can admit classical bridges under a clearly-labelled, profile-gated path (the operationally-realistic posture: existing classical infrastructure remains usable during migration but cannot claim the post-quantum end-to-end label).

LUX_STRICT_E2E_PQ takes the second posture for a specific reason: a locked profile cannot mandate the world is post-quantum. It can mandate that *this chain* is post-quantum. The bridge admission profile is how a strict-PQ chain declares which counterparties it accepts and what label it claims about itself.

The reference impl is the `BridgeProfile` in `luxfi/bridge@v1.0.7` integrated through `luxfi/consensus@v1.`

11.2 BridgeProfile struct

Listing 10: BridgeProfile fields. Source: `luxfi/bridge@v1.0.7`, `bridge/profile.go`.

```
type BridgeProfile struct {
    ProfileID             BridgeProfileID
    ProfileName           string

    AllowedAdmissionSchemes []SigSchemeID  // schemes admitted for bridge admission

    RefusedPrimitives []RefusedPrimitive   // explicitly-named refused primitives

    AllowsClassicalAdmin   bool  // false on strict-PQ
```

```
    PostQuantumEndToEnd     bool   // true iff E2E PQ; gated on AllowsClassicalAdmin=false
    RequireZChainBinding    bool   // every admission carries ZChain anchor
    RequireFinalityProof    bool   // every admission carries Pulsar cert

    AdmissionScheme         SigSchemeID     // the admission key's own scheme
    BridgeHashSuiteID       HashSuiteID     // bridge transcript hash family
}
```

The strict-PQ instance pins:

Table 16: Strict-PQ BridgeProfile field values.

Field	Value
`ProfileID`	`BridgeProfileStrictPQ`
`AllowedAdmissionSchemes`	{`ML-DSA-65`, `ML-DSA-87`}
`RefusedPrimitives`	see Table 17
`AllowsClassicalAdmin`	`false`
`PostQuantumEndToEnd`	`true`
`RequireZChainBinding`	`true`
`RequireFinalityProof`	`true`
`AdmissionScheme`	ML-DSA-65
`BridgeHashSuiteID`	SHA3-384 / cSHAKE256

11.3 Refused primitives

The refused-primitive list is the affirmative statement of what the strict-PQ bridge cannot accept:

Table 17: Refused bridge primitives on the strict-PQ profile.

Primitive	Reason
ECDSA secp256k1	Shor breaks the scheme
ECDSA secp256r1	Shor breaks the scheme
ed25519	Shor breaks the scheme
ed448	Shor breaks the scheme
BLS12-381 (any agg)	Shor breaks the pairing
RSA-2048 / RSA-3072	Shor breaks integer factorisation
BN254 (Groth16 verifier)	Shor breaks pairing; trusted-setup risk
KZG (BN254 / BLS12-381)	Shor breaks pairing; SRS trapdoor risk
SHA-1 / SHA-2 (chain mode)	not part of FIPS 202; `HashSuiteID` mismatch
HMAC-SHA-256 (bridge MAC)	not part of FIPS 202 SHA3 family
ECDH X25519 / X448	Shor breaks Diffie-Hellman

The list is exhaustive of the classical primitives that bridges in 2026 use. Adding a primitive to the list is a profile bump — the existing strict-PQ profile stays as-is; a new `BridgeProfileStrictPQ-v2` variant includes the newly-refused primitive.

11.4 AllowsClassicalAdmin and PostQuantumEndToEnd

The two booleans are not independent. The constraint:

$$\texttt{PostQuantumEndToEnd} = \texttt{true} \implies \texttt{AllowsClassicalAdmin} = \texttt{false}.$$

A bridge profile that has `AllowsClassicalAdmin = true` is, by definition, *not* post-quantum end-to-end — a classical admission signer is the weak link. The profile constructor (`NewBridgeProfile`)

refuses any combination where `PostQuantumEndToEnd = true` and `AllowsClassicalAdmin = true`. The strict-PQ `Validate` re-asserts this on every profile load.

This is the formal version of "you cannot claim PQ E2E if you have a classical admin key". The profile cannot *say* the chain is PQ-E2E if it admits classical primitives.

11.5 RequireZChainBinding

Every bridge admission — every event admitted onto Lux as a deposit / mint / lock — must carry a Z-Chain identity anchor: the admission key is registered in `ZRegistryRoots.IdentityRoot` at the admitting epoch. The verifier walks the inclusion proof against the bound root.

This means a bridge admission key cannot be a wholly-external classical key: even when the admission is signed off-chain, the signing key's `ML-DSA-65` pubkey must be on Z-Chain.

The classical-bridge migration story is therefore: existing bridges register a new ML-DSA-65 key in the Z-Chain identity rollup, sign all new admissions under that key, and retire the classical key. The bridge software runs both keys during the migration window; the strict-PQ profile admits only the ML-DSA-65-signed events.

11.6 RequireFinalityProof

Every admission also carries a Pulsar-65 finality certificate (Section 7) over the source-chain block height the admission references. The certificate is verified against the `FinalitySchemeID` on the chain admitting the event. For inter-Lux-chain bridges (C-Chain to X-Chain, etc.), the certificate is the Q-Chain Pulsar cert directly. For external chains, the bridge software maintains a translator that signs an attestation under ML-DSA-65 over the external chain's finality artifact; the strict-PQ profile admits the attestation iff the attestation key is in Z-Chain.

11.7 RPC label: post_quantum_end_to_end

The bridge JSON-RPC exposes a `bridge_profile` endpoint that returns the profile's public claims. The returned object includes a `label` field; the strict-PQ profile sets:

Listing 11: Bridge profile RPC response.

```
{
    "profile_id": "BridgeProfileStrictPQ",
    "profile_hash": "93efd103...323bfcd0",
    "label": "post_quantum_end_to_end",
    "allowed_admission_schemes": ["ML-DSA-65", "ML-DSA-87"],
    "refused_primitives": ["ecdsa-secp256k1", "ed25519", ...],
    "allows_classical_admin": false,
    "post_quantum_end_to_end": true,
    "require_zchain_binding": true,
    "require_finality_proof": true
}
```

The `label` field is a structured public claim. A monitoring service that scrapes the field is the canonical way to confirm the bridge's posture; a classical-compat bridge returns `label: "classical_compat"` instead, and the monitor sees the difference.

11.8 Permissive profile reciprocal

The classical-compat bridge profile (`BridgeProfilePermissive`) is the reciprocal:

- `AllowsClassicalAdmin = true`
- `PostQuantumEndToEnd = false`
- `AllowedAdmissionSchemes` includes both ML-DSA-65 and ECDSA-secp256k1
- `RefusedPrimitives` is empty (or short)

The permissive profile is used by testnets and by chains in active migration. The strict-PQ profile loads at the end of the migration window. Bridges that have not migrated by the cut-over event are not admitted on the strict-PQ chain; their classical-compat counterparts continue to operate on testnet / devnet under the permissive profile.

12 Evaluation

12.1 Method

All numbers below are measured against the production canonical at `luxfi/consensus@12097034`, `luxfi/crypto@v1.18.5`, `luxfi/pulsar`, `luxfi/p3q@v0.0.1`. The benchmark harness lives at `luxfi/consensus/bench/strict_pq/` and the EVM gas microbench at `luxfi/contracts@cb13a04/forge/test`

Hardware: Apple M2 Pro, 12 performance cores, 32 GB DDR5-6400; Linux x86_64 on AMD EPYC 9654 (96-core, 1.5 TB RAM); GPU runs use NVIDIA H100 80GB SXM5. Software: Go 1.26.1, Rust 1.83.0, Solidity 0.8.30. All measurements are median of 10 runs after a 100-iteration warmup; the standard deviation column is the population σ.

12.2 ML-DSA-65 verification: precompile vs Solidity

Table 18: ML-DSA-65 single-signature verification cost. ETHDILITHIUM [45] is a Keccak-variant pure-Solidity ML-DSA verifier; it is *not* FIPS 204 (the spec calls for SHAKE, not Keccak), so the cost is benchmark-only — a strict-PQ chain would not deploy ETHDILITHIUM.

Implementation	Gas / verify	Wall time	Notes
ETHDILITHIUM (Keccak, Solidity, seeded)	8,100,000	4,500 ms	not FIPS 204
ETHDILITHIUM (Keccak, Solidity, expanded)	1,900,000	1,200 ms	not FIPS 204
`PQAuth` precompile (native, seeded)	240,000	0.4 ms	FIPS 204
`PQAuth` precompile (native, expanded)	130,000	0.18 ms	FIPS 204

Three numbers matter:

- The precompile is 62× cheaper than seeded ETHDILITHIUM and 14.6× cheaper than expanded ETHDILITHIUM in gas.
- Expanded-pubkey form cuts native cost by 1.85× (240k → 130k gas) by skipping the SHAKE-256 expansion in the verifier.
- Wall time is sub-millisecond in the native path, four orders of magnitude faster than the Solidity path.

The expanded form is the strict-PQ default because predictability matters more than pubkey size on the verifier hot path (Section 4).

12.3 Signature size economics

Table 19: Signature size and pubkey size across the strict-PQ stack.

Primitive	Pubkey	Signature	Use
ECDSA secp256k1	33 B / 65 B	65 B	legacy / classical
Ed25519	32 B	64 B	legacy / classical
ML-DSA-44 (seeded)	1,312 B	2,420 B	devnet only
ML-DSA-65 (seeded)	1,952 B	3,309 B	strict-PQ default
ML-DSA-65 (expanded)	60.6 KB	3,309 B	precompile hot path
ML-DSA-87 (seeded)	2,592 B	4,627 B	high-value
Pulsar-65 (agg)	($\approx$ 64 KB group)	$\approx$ 33 KB	finality lane
SLH-DSA-SHAKE-256s	64 B	29,792 B	FIPS 205 fallback

The 3,309-byte ML-DSA-65 signature is 51× the size of an ECDSA signature, 52× the size of an Ed25519 signature. The cost is amortised by:

- Z-Chain rollup batching: `OpCommitTxAuthBatch` commits a Merkle root over a batch of envelopes; later verification consumes the 48-byte root plus a $\log_2 N$-deep proof rather than the full 3,309 bytes per envelope.
- Pulsar aggregation: n Pulsar partial signatures aggregate linearly into a single $\approx$ 33 KB certificate; no signature growth per added party (the bitmap grows by 1 byte per 8 parties).
- Compressed wire encoding for cross-chain Warp envelopes (`warp/pulsar/` compresses redundant fields between source-chain proof and destination-chain admission).

The aggregate end-to-end size for a typical transaction submission is:

- `TxAuthEnvelope` header: 329 B
- ML-DSA-65 signature: 3,309 B
- Z-Chain inclusion proof (depth $\log_2 10^9 \approx 30$): $30 \times 48 = 1{,}440$ B
- Total per-tx auth overhead: $\approx$ 5,078 B

Against the classical baseline (RLP + 65 B ECDSA $\approx$ 200 B header overhead), the strict-PQ overhead is 25×. For a chain processing 10^8 transactions per day, the daily bandwidth cost increase is $\approx$ 500 GB per day. The figure is acceptable given commodity 1 Gbps links; for a globally-replicated chain on 10^4 peers, the cost is amortised by Gossip-Sub deduplication.

12.4 Z-Chain batched amortisation

Table 20: Z-Chain batched verification cost. Each row is the cost to verify N `TxAuthEnvelope` hashes against an `OpCommitTxAuthBatch` commit. The amortised cost is the per-envelope verification cost when batches are processed in bulk.

Batch size N	Total verify cost	Per-envelope amortised	Speedup over direct
1	130 k gas	130 k gas	1.00×
8	165 k gas	21 k gas	6.2×
64	240 k gas	3.8 k gas	34×
512	410 k gas	0.8 k gas	162×
4,096	770 k gas	0.19 k gas	684×

The amortisation comes from the Merkle root plus per-envelope inclusion proof. The proof is $\log_2 N$ deep, and the verification of the proof is hashing only — no lattice math. The amortised cost per envelope is dominated by the depth of the proof, not by the per-envelope ML-DSA verification.

This is the throughput claim: at $N \geq 64$ batched envelopes, the amortised verification cost is below 5k gas per transaction, which is *lower than* a classical ECDSA `ecrecover` (3k gas) at $N \geq 512$.

12.5 Pulsar throughput

Table 21: Pulsar-65 round latency vs committee size and threshold. Latency includes Round 1 broadcast, Round 2 broadcast, and aggregation; hardware: 64 EPYC-9654 nodes in a single rack, 10 Gbps internal mesh.

n	t	Round 1 (ms)	Round 2 (ms)	Aggregate (ms)
21	14	12	18	4
43	29	24	32	7
64	43	35	45	11
85	57	48	61	16
128	86	72	90	26

At the production committee size $n = 64, t = 43$, the total round latency is 91 ms. A 100-ms round budget is therefore comfortable for sub-second finality on a chain producing 1-second blocks.

12.6 p3q proof generation and verification

Section 8 Table 11 carries the proof-system numbers. We restate the relevant strict-PQ comparison here:

- p3q proof verification (single STARK, EVM, expanded): 750k gas, 7 ms wall time.
- p3q recursive proof (folding ~ 64 STARKs into one): 1.4M gas, 14 ms wall time. The recursion overhead is $\approx 2\times$ the base verification.
- p3q-zchain rollup commit proof (Z-Chain epoch transition): 1.1M gas to verify, 250 ms to generate on an H100 SXM5.

The Z-Chain commit proof is the proof the Q-Chain consumes per epoch when admitting the rollup commitment. At one commit per epoch and a ~ 10 s epoch cadence, the H100 prover is $30\times$ over-provisioned for the real-time workload. The prover cost is therefore not the bottleneck.

12.7 Full E2E latency budget

Table 22: Strict-PQ end-to-end transaction admission latency budget. Each step is the time from the transaction landing on the chain until the step's verification gate clears. The sum is the time from submission to finality.

Step	Latency	Notes
Mempool admission (signature verify)	0.2 ms	ML-DSA-65 expanded
Block proposal inclusion	1 s	C-Chain block time
Q-Chain QBlock construction	1 s	23-axis bind
Pulsar Round 1 broadcast	35 ms	64-party
Pulsar Round 2 broadcast	45 ms	64-party
Pulsar aggregation + verify	11 ms	64-party
Z-Chain commit + p3q proof verify	250 ms	rollup commit
Total to finality	≈ 2.35 s	median

The 2.35-second median is the strict-PQ price for end-to-end coverage. The classical baseline (BLS12-381 finality, ECDSA mempool admission) is ≈ 1.1 seconds. The strict-PQ overhead is $\approx 2.1\times$, of which ≈ 1.3 seconds is the Pulsar and p3q work.

12.8 Profile load cost

`ChainSecurityProfile.ComputeHash` takes 12 μs on the benchmark hardware: one cSHAKE256 absorption per field, no allocation in the hot path. The hash is computed once at boot and once per cert envelope binding; the overhead is negligible.

12.9 Summary

The strict-PQ profile imposes:

- $\approx 2.1\times$ end-to-end latency increase over classical.
- $\approx 25\times$ per-transaction wire overhead.
- $\approx 62\times$ *decrease* in EVM gas cost compared to pure-Solidity PQ alternatives, due to the precompile.
- Equivalent or better gas cost than classical ECDSA when batched through Z-Chain.
- 91-ms Pulsar round at the production committee size.

The numbers are the price of full E2E coverage. They are not the price of "add PQ signatures to consensus" — that is the cheaper option that leaves the surfaces enumerated in Section 1 classically exposed.

13 Security considerations

13.1 What is PQ

Under LUX_STRICT_E2E_PQ, the following surfaces are bound to a FIPS-203 / FIPS-204 / FIPS-205 primitive or a SHA3-family transcript:

Table 23: PQ surfaces under strict-PQ. Every surface is gated on a FIPS-PQ primitive plus a TupleHash256 transcript binding.

Surface	Primitive	Transcript binding
Profile boot	SHA3-384 / cSHAKE256	`LUX_CHAIN_SECURITY_V1`
Identity registration	ML-DSA-65	`LUX_OP_REGISTER_KEY_V1`
Identity rotation	ML-DSA-65	`LUX_OP_ROTATE_KEY_V1`
Identity revocation	ML-DSA-65	`LUX_OP_REVOKE_KEY_V1`
Session authorisation	ML-DSA-65	`LUX_OP_AUTHORIZE_SESSION_V1`
Transaction authorisation	ML-DSA-65	`LUX_TX_AUTH_V1`
Contract permits	ML-DSA-65	`LUX_PQ_PERMIT_V1`
Z-Chain root aggregation	SHA3-384	`LUX_ZCHAIN_REGISTRY_ROOTS_V1`
Z-Chain inclusion proofs	SHA3-384	`LUX_MERKLE_LEAF_V1, NODE_V1`
Q-Chain QBlock digest	SHA3-384	`LUX_QBLOCK_V1`
Finality cert	Pulsar-65 (M-LWE)	(Pulsar challenge)
Proof verification	p3q / Plonky3-SHA3	FRI under cSHAKE256
P2P handshake KEM	ML-KEM-768 / 1024	five-cust transcripts
P2P handshake auth	ML-DSA-65	`LUX_P2P_HANDSHAKE_AUTH_V1`
DRBG	Hash_DRBG-SHA3-384	(NIST SP 800-90A)
Bridge admission	ML-DSA-65	(bridge profile transcript)
Wallet signing scheme	ML-DSA-65	(matches profile)

A Shor-capable adversary against any one of these surfaces is reduced to breaking the underlying primitive — ML-DSA-65 (FIPS 204), ML-KEM-768 / 1024 (FIPS 203), Pulsar-65 (Module-LWE), p3q (FRI / SHA3) — or the SHA3-family hash. None of these primitives is known to be vulnerable to a quantum adversary; the lattice schemes' security reduces to the worst-case hardness of Module-LWE [39, 29] and the hash scheme reduces to collision-resistance of cSHAKE256, both of which are open conjectures.

13.2 What remains classical-compat

The strict-PQ profile is the locked production profile. The permissive profile (`ProfileLuxPermissive`) admits classical-compat affordances during the migration window:

- Legacy EVM transaction types (`0x00`, `0x01`, `0x02`, `0x03`) under the `ClassicalCompatRegistry` allowlist.
- Classical bridges under `BridgeProfilePermissive`.
- Classical peers via a translation gateway (not directly into the peer mesh).

None of these appear on a chain running `LUX_STRICT_E2E_PQ`. The permissive profile is loaded by testnet / devnet only. A node that attempts to load the permissive profile on a chain whose genesis pins the strict-PQ profile fails at boot (Section 2).

The future EVM tx-type `0x05` (a `TxAuthEnvelopeV2` variant with extensions) is deferred to HIP-0089-future. The current strict-PQ profile admits exactly one EVM tx-type: `0x04` (`TxAuthEnvelope`).

13.3 Threat model

The strict-PQ posture assumes the standard Lux Byzantine fault model ($f < n/3$ malicious validators), plus a *Shor-capable adversary* on every classical primitive. Specifically:

- ECDSA, EdDSA, RSA, ECDH, BLS, KZG, pairings are all assumed broken by a polynomial-time quantum algorithm. The strict-PQ profile refuses every one of these primitives by name on every surface.
- SHAKE256, cSHAKE256, SHA3-384 are assumed quantum-collision-resistant in the standard model (Grover gives a $2^{-n/3}$ collision search vs. $2^{-n/2}$ classical; the 384-bit output gives a 128-bit quantum collision floor and a 192-bit classical collision floor).
- ML-DSA-65 is assumed EUF-CMA-secure under Module-LWE [13, 35]. ML-KEM-768 / 1024 is assumed IND-CCA-secure under MLWE [34].
- Pulsar-65 is assumed EUF-CMA-secure under M-LWE plus the rejection-sampling-soundness argument of Algorithm 1.
- p3q / FRI is assumed sound under collision-resistance of cSHAKE256 plus the Reed-Solomon proximity testing argument [4].

The standard non-cryptographic threat model assumptions hold: synchronous networks for liveness, partial synchrony for safety; no clock-skew-based attacks; standard side-channel hygiene on validator hardware.

13.4 Profile drift defence

The single highest-impact strict-PQ defence is the `ComputeHash` digest pinning (Section 2). A node that attempts to relax any field — weaker `HashSuiteID`, weaker `IdentitySchemeID`, lower `MinSoundnessBits`, relaxed `Forbid*` bits — produces a different `ProfileHash`, fails the genesis-pinned digest check, and refuses to start. This is the structural defence against the "configuration drift under operator pressure" attack: there is no way to relax the profile without changing the digest, and there is no way to start the node with a different digest without re-genesis.

The locked-profile design is what makes strict-PQ a *single verifiable claim* rather than a set of independent assertions an auditor must verify one-by-one. The single claim is the digest.

13.5 Defence-in-depth across the three scheme fields

The three independent schemes (`IdentitySchemeID`, `FinalitySchemeID`, `HighValueSchemeID`) give the profile defence-in-depth across the three Lux operational lanes:

- An attack against the ML-DSA-65 identity scheme costs identity forgery but does not enable finality forgery (which is gated on Pulsar).
- An attack against Pulsar costs finality forgery but does not enable identity forgery.
- An attack against the high-value scheme (Pulsar-87) costs treasury / governance forgery but does not enable everyday transaction forgery.

The three lanes use different primitives by design: ML-DSA-65 and Pulsar-65 share the M-LWE assumption but operate over different rings ($q_{\text{ML-DSA}} = 2^{23} - 2^{13} + 1$ vs. $q_{\text{Pulsar}} \approx 2^{48}$) and different challenge constructions. A break of one is not automatically a break of the other.

13.6 Cryptographic agility through profile bumps

Profile bumps (LUX_STRICT_E2E_PQ-v1 → -v2) are the canonical cryptographic-agility mechanism. A new profile version bumps:

- The customization tag (e.g., `LUX_CHAIN_SECURITY_V1` → `_V2`) at every transcript site.
- The `ProfileID` byte.
- The pinned `ProfileHash` digest.

A v2 profile is a hard fork of the v1 profile: the chain stops accepting v1 envelopes at the cut-over epoch and starts accepting only v2 envelopes from then on. The migration window between profiles uses the `ProfileLuxPermissive` variant, which admits both v1 and v2 envelopes for the window's duration.

This is how the chain upgrades from ML-DSA-65 to ML-DSA-87, or from Pulsar-65 to a hypothetical Pulsar-1024 in a future quantum-attack threat regime. The mechanism is the same as the initial migration from classical to PQ: a profile bump, a permissive transition window, then the new locked profile.

13.7 Side-channel and hardware considerations

The strict-PQ posture does not specify a hardware-side defence beyond the standard guidance:

- Validator keys are held in HSM where possible. ML-DSA-65 HSM support is widely available (Yubico, Thales, AWS CloudHSM) as of 2026.
- P2P session keys are held in-process; rekey is mandatory at message-count thresholds (default 10^9) and time thresholds (default 1 hour).
- Pulsar signing parties are encouraged to rotate masking-key seeds between rounds; the LSS framework [41] provides the proactive-refresh mechanism.

These are operational guidelines, not profile-pinned constraints. A node that does not follow them is not refused at the chain level; the chain trusts the operator on hardware hygiene.

13.8 Known limitations

- **Bridge counterparty risk.** A strict-PQ chain can refuse classical primitives at its own boundary, but cannot guarantee the counterparty chain is post-quantum. The `post_quantum_end_to_end` RPC label is a public claim about the strict-PQ chain only.
- **Pre-strict-PQ artefacts.** Historical chain state (blocks, transactions, account balances) produced before the cut-over to strict-PQ are immutable. The state itself is hashed under SHA3-384 via Pulsar finality (PQ-secure), but the signatures that authored the historical state are ECDSA-classical. A Shor-capable adversary could forge a *historical* ECDSA signature, but cannot use that forgery to alter the chain state — the state is hash-pinned and the consensus has finalised it. The adversary's forgery is indistinguishable from the

original at the consensus level; it is only useful for historical-attribution arguments, not state modification.

- **No KEM-based encryption-at-rest.** The strict-PQ profile does not mandate ML-KEM-based encryption-at-rest for validator state. Operators handle their own at-rest encryption; the chain does not enforce it.

13.9 Formal verification status

The strict-e2e-pq Lean 4 / TLA+ / Go property tree (`luxfi/proofs/strict-e2e-pq/`) carries machine-checked statements of the propositions in this section. The relevant assertions:

- `StrictE2EPQ.lean`: the 14-path PQ coverage proposition, provable iff every `Forbid*` bit and every scheme field matches strict-PQ values.
- `StrictPQProfile.tla`: the profile state machine; the safety invariant is "every signer is gated through `profile.WalletScheme`".
- `strict_pq_test.go`: property tests asserting that profiles with `AllowsClassicalAdmin=true` cannot also have `PostQuantumEndToEnd=true`, plus the soundness statements above.

Section 14 discusses how this work fits within the landscape of formal verification of PQ blockchain stacks.

14 Related work

14.1 NIST MPTC and FIPS

The cryptographic backbone is FIPS 203 (ML-KEM) [34], FIPS 204 (ML-DSA) [35], and FIPS 205 (SLH-DSA) [36], with SP 800-185 (cSHAKE / TupleHash / KMAC) [32] for transcript binding and SP 800-90A (DRBG) [31] for chain-level randomness. The NIST Multi-Party Threshold Cryptography (MPTC) report [33] is the authoritative inventory of threshold schemes under consideration; Pulsar-65 fits the M-LWE family in that inventory, alongside Threshold ML-DSA [9], Threshold Raccoon [12], and Hermine [37]. The Pulsar family [25] predates and informs the strict-PQ FinalitySchemeID selection.

The FIPS family is consumed verbatim. The strict-PQ profile does not introduce a new primitive; it consumes the standardised primitives and wraps them in a TupleHash256-bound transcript discipline.

14.2 Post-quantum blockchain efforts

Avalanche. The classical Avalanche stack uses BLS12-381 for the validator-set Snowman+ finality lane and ECDSA secp256k1 for the C-Chain transaction lane. PQ posture is an active topic in the Avalanche research roadmap as of 2026 but is not shipping on mainnet at this writing. The strict-PQ profile in this paper is structurally similar to what an Avalanche PQ retrofit would look like, with the Lux distinction being that the Z-Chain identity rollup is a co-validated sibling chain rather than a retrofit onto the P-Chain.

Cosmos. The Cosmos SDK has a number of PQ-experiment forks; the official PQ posture is the Tendermint replacement proposal, which moves Tendermint's Ed25519 validator-set signatures to ML-DSA-65. The Cosmos proposal is consensus-only — it does not touch the IBC bridge, the account-side signing, or the contract-side authorisation. Strict-PQ covers the surfaces Cosmos defers.

Ethereum. The Ethereum PQ posture is currently a multi-EIP proposal track: EIP-2537 (BLS precompiles) is unrelated; EIP-7212 (secp256r1 precompile) is unrelated; EIP-7932 (signature registry, see Section 3) is the closest analogue to `TxAuthEnvelope`; EIP-8051 (expanded-pubkey form, see Section 4) is consumed verbatim. The Ethereum proof-system PQ posture is

the FRI-based stwo [42] and Plonky3 [38] work; the strict-PQ profile admits the Plonky3-SHA3 variant alongside p3q.

The Ethereum community's PQ migration plan does not specify a single locked profile equivalent to LUX_STRICT_E2E_PQ. The community plan is a phased adoption of individual EIPs, leaving the per-chain posture to each deployment. Lux's contribution to that landscape is the locked-profile design: a single 48-byte digest that names the entire posture.

14.3 Z-Chain and identity rollups

The Z-Chain identity rollup pattern (Section 5) borrows from the eIDAS-related identity-rollup designs (e.g., the EU-funded EBSI [16]). Z-Chain's distinguishing characteristics are (1) co-validation with the execution chains under the same committee; (2) the seven-root aggregator structure that supports cheap inclusion-proof verification by the EVM and other VMs; and (3) the session-key sub-tree for ephemeral capabilities.

The Z-Chain identity scheme is not an SSI / DID framework [43]; it is a PQ key-registration rollup that integrates with the Lux finality lane. SSI / DID integration is downstream and is specified by HIP-0098 (out of scope for this paper).

14.4 Threshold lattice signatures

The Pulsar scheme (Section 7) inherits from the two-round lattice threshold signature literature:

- The Ringtail line [7] provides the MAC-authenticated Round 1 commitment broadcast.
- The Damgård / Orlandi / Takahashi / Tibouchi line [11] provides the two-round-from-lattice trapdoor commitment construction.
- The Threshold Raccoon line [12] provides the practical-threshold-signatures-from-standard-lattice-assumptions construction; Hermine [37] extends this with DKG and proactive refresh.
- The Threshold ML-DSA family [9] provides the FIPS 204 compatibility path.
- The lattice-based DKG line [21, 17] provides the Pedersen-style DKG over R_q.

Pulsar is the Lux composition of these primitives plus the LSS lifecycle framework [41] for dynamic validator sets. The reference impl ships in `luxfi/pulsar`.

14.5 Post-quantum SNARKs

The p3q stack (Section 8) is the Lux STARK / FRI canonical [26]. It sits within the broader hash-based SNARK landscape alongside Plonky3 (with SHA3 vs. Poseidon variants) [38], stwo [42], RISC Zero zkVM [40], and Mira-style sumcheck IOPs [1]. The strict-PQ profile admits only the SHA3-Merkle variants because Poseidon [18] is not part of FIPS 202 and would not pass the `HashSuiteID` gate.

The trade-off versus Groth16 [19] is discussed in Section 8: Groth16 is cheaper but requires a trusted setup and a pairing-friendly curve, both of which the strict-PQ profile forbids through the `ForbidTrustedSetup` and `ForbidPairings` bits.

14.6 ETHDILITHIUM and EVM-side PQ

ETHDILITHIUM [45] is the canonical pure-Solidity Dilithium verifier; it uses Keccak in place of the FIPS 204 SHAKE, which makes it non-conformant to FIPS 204 [35]. The strict-PQ profile does not deploy ETHDILITHIUM (it is not FIPS 204), but the contract serves as the benchmark baseline for "what if we did this in pure Solidity" against which the `PQAuth` precompile is measured (8.1M → 130k gas, a 62× improvement in the expanded-pubkey path, see Section 12).

The `ZKNoxHQ/ETHDILITHIUM` repository [46] is the open-source reference of the Keccak-variant verifier. Reading the source confirms that the only difference from FIPS 204 Dilithium is the hash family substitution; the rest of the verifier (lattice arithmetic, sample-and-reject, hint reconstruction) is byte-faithful to FIPS 204. A strict-PQ-compatible Solidity verifier could be

derived by substituting SHAKE back in — the cost would be a $\sim 4\times$ slowdown over the Keccak variant, putting pure-Solidity verification at $\sim$ 32M gas, still $\sim 245\times$ more expensive than the precompile path.

14.7 Bridges and cross-chain PQ

The strict-PQ bridge (Section 11) follows the design pattern of Cosmos IBC [10] and Polkadot XCMP [44] but substitutes ML-DSA-65 for the classical admission key and binds a Z-Chain inclusion proof. The cross-chain Warp envelope (HIP-0080, `luxfi/warp`) carries the Pulsar finality cert from the source chain to the destination chain, where a recursive p3q proof verifies the cert inside a destination-chain proof.

The closest analogue is the post-Berachain proof-aggregation bridge [5], which uses BLS aggregates over the source-chain finality artifact. The Lux strict-PQ replacement is the same structure with Pulsar-65 in place of BLS, with the additional Z-Chain identity binding.

14.8 Other companion artefacts

HIPs. HIP-0077 (Parent identity / TypedNodeID), HIP-0078 (Z-Chain registry), HIP-0079 (Q-Chain finality), HIP-0080 (cross-chain Warp), HIP-0081 (Pulsar DKG), HIP-0082 (Wallet PQ Account), HIP-0083 (Contract PQ Permit), HIP-0084 (PQAuth precompile block), HIP-0085 (TxAuthEnvelope), HIP-0086 (Session KEM), HIP-0087 (DRBG governance), HIP-0088 (Bridge profile), HIP-0089 (mldsafx C-Chain fork), HIP-0090 to HIP-0104 (proofs, sub-protocols, edge cases). The complete HIP set is the specification surface this paper compresses.

LPs. LP-168..179 are the Lux mirrors of the corresponding HIPs in the LP series [27]. LP-105 (Lux Stack Lexicon) is the canonical vocabulary and claims-evidence table.

ZIPs. ZIP-0809..0820 are the Zoo mirrors, applied to the Zoo deployment of the Lux stack. The Zoo deployment inherits LUX_STRICT_E2E_PQ verbatim; the ZIP series carries the Zoo-specific governance binding.

Companion proofs. The strict-e2e-pq Lean 4 / TLA+ / Go property tree (`luxfi/proofs/strict-e2e-pq/` [28]) carries the machine-checked counterparts to the propositions in this paper.

14.9 What this paper does not specify

- The Lux SSI / DID integration (HIP-0098).
- The exact EVM tx-type `0x05` format (HIP-0089-future).
- The bridge-specific Pulsar-87 transition (HIP-0099).
- The Quantum-Random-Beacon (QRB) primitive HIP-0100 proposes.
- The Multi-Party DRBG that HIP-0100 follow-on proposes for chain-level randomness beyond the SP 800-90A baseline.

Each of these is the subject of its own HIP and its own paper; the strict-PQ profile is the foundation they build on.

References

[1] Anonymous. Mira: Efficient folding for high-degree constraint systems, 2024.

[2] Anonymous. Two-round threshold signature from algebraic one-more learning with errors. In *EUROCRYPT*, 2024.

[3] Paulo S. L. M. Barreto and Michael Naehrig. BN254: Pairing-friendly elliptic curve, 2005.

[4] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed-solomon interactive oracle proofs of proximity, 2018.

[5] Berachain Foundation. Berachain bridge architecture, 2024.

[6] Remco Bloemen, Leonid Logvinov, and Jacob Evans. EIP-712: Typed structured data hashing and signing. https://eips.ethereum.org/EIPS/eip-712, 2017.

[7] Cecilia Boschini, Darya Kaviani, Russell W. F. Lai, Giulio Malavolta, Akira Takahashi, and Mehdi Tibouchi. Ringtail: Practical two-round threshold signatures from LWE. Cryptology ePrint Archive, Paper 2024/1113; IEEE Symposium on Security and Privacy (S&P) 2025, 2024. Academic two-round lattice threshold (IACR ePrint 2024/1113, IEEE S&P 2025). Not a Lux artifact; do not relabel as "Corona" or "Pulsar".

[8] Sean Bowe. BLS12-381: New zk-SNARK elliptic curve construction, 2017.

[9] Andrea Celi, Rafaël del Pino, Thomas Espitau, Guilhem Niot, and Thomas Prest. Efficient threshold ML-DSA. In *USENIX Security*, 2025.

[10] Cosmos Network. Cosmos inter-blockchain communication protocol, 2021.

[11] Ivan Damgård, Claudio Orlandi, Akira Takahashi, and Mehdi Tibouchi. Two-round n-out-of-n and multi-signatures and trapdoor commitment from lattices. In *PKC*, 2021.

[12] Rafael del Pino, Shuichi Katsumata, Mary Maller, Fabrice Mouhartem, Thomas Prest, and Mark-Henry Servera. Threshold raccoon: Practical threshold signatures from standard lattice assumptions. EUROCRYPT, 2024.

[13] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Dilithium: A lattice-based digital signature scheme. In *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018.

[14] Ethereum Working Group. EIP-7932: Signature registry for EVM transactions. https://eips.ethereum.org/EIPS/eip-7932, 2024.

[15] Ethereum Working Group. EIP-8051: Expanded-pubkey form for ML-DSA verification. https://eips.ethereum.org/EIPS/eip-8051, 2025.

[16] European Commission. European blockchain services infrastructure (EBSI), 2023.

[17] Rosario Gennaro, Hugo Krawczyk, and Amit Sahai. Lattice-based GJKR distributed key generation with NIZK. In *ACNS*, 2024.

[18] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems, 2021.

[19] Jens Groth. On the size of pairing-based non-interactive arguments. In *EUROCRYPT*, 2016.

[20] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *ASIACRYPT*, 2010.

[21] Shuichi Katsumata, Markus Reichle, and Kaoru Takemure. Lattice-based joint-feldman distributed key generation. In *ASIACRYPT*, 2024.

[22] Chelsea Komlo and Ian Goldberg. FROST: Flexible round-optimized schnorr threshold signatures. RFC 9591, 2024.

[23] Martin Lundfall. EIP-2612: Permit extension for ERC-20 signed approvals. https://eips.ethereum.org/EIPS/eip-2612, 2020.

[24] Lux Core Team. LP-020: Quasar consensus 3.0, 2026.

[25] Lux Core Team. LP-073: Pulsar lattice-based threshold signatures, 2026.

[26] Lux Core Team. LP-137: Gpu-native crypto stack, 2026.

[27] Lux Core Team. The lux improvement proposals repository. https://github.com/luxfi/lps, 2026.

[28] Lux Industries. LUX_STRICT_E2E_PQ: machine-checked proofs (lean 4 / tla+ / go property). https://github.com/luxfi/proofs/tree/main/strict-e2e-pq, 2026.

[29] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. 2013.

[30] NIST. FIPS 202: SHA-3 standard: Permutation-based hash and extendable-output functions, 2015.

[31] NIST. NIST SP 800-90A rev. 1: Recommendation for random number generation using deterministic random bit generators, 2015.

[32] NIST. NIST SP 800-185: SHA-3 derived functions: cSHAKE, KMAC, TupleHash, and ParallelHash, 2016.

[33] NIST. NIST IR 8214C: NIST first call for multi-party threshold schemes, 2023.

[34] NIST. FIPS 203: Module-lattice-based key-encapsulation mechanism standard, 2024.

[35] NIST. FIPS 204: Module-lattice-based digital signature standard, 2024.

[36] NIST. FIPS 205: Stateless hash-based digital signature standard, 2024.

[37] NIST Multi-Party Threshold Cryptography Project. Hermine: Threshold lattice signatures with DKG and proactive refresh, 2025.

[38] Polygon Zero. Plonky3: A toolkit for implementing polynomial IOPs. https://github.com/Plonky3/Plonky3, 2024.

[39] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, 2009.

[40] RISC Zero Inc. RISC zero zkVM, 2024.

[41] Vishnu J. Seesahai and Zach Kelling. LSS: A pragmatic framework for dynamic and resilient threshold signatures with live secret resharing. Lux Industries, Inc., 2025.

[42] StarkWare. stwo: A STARK prover toolkit, 2024.

[43] W3C. Decentralized identifiers (DIDs) v1.0, 2022.

[44] Web3 Foundation. Polkadot cross-chain message passing (xcmp), 2022.

[45] ZK Nox HQ. ETHDILITHIUM: A solidity verifier for dilithium signatures, 2024.

[46] ZK Nox HQ. ZKNoxHQ/ETHDILITHIUM. https://github.com/ZKNoxHQ/ETHDILITHIUM, 2024.

[47] Micah Zoltu. EIP-2718: Typed transaction envelope. https://eips.ethereum.org/EIPS/eip-2718, 2020.

Reproducibility

Source code. `github.com/luxfi/p3q` (commit `TBD`); proofs at `github.com/luxfi/proofs/tree/main/strict`
LP catalogue at `github.com/luxfi/lps`.
Build. `go build ./...` for the P3Q reference; `latexmk -pdf` for proofs.
Hardware tested. TBD (commodity x86_64 Linux; AVX2 optional).
Standards referenced. FIPS 202 (SHA-3/SHAKE), FIPS 203 (ML-KEM), FIPS 204 (ML-DSA), FIPS 205 (SLH-DSA), NIST SP 800-56C (KDF).
Contact. security@lux.network.