Lux MPTC Serialization Specification

Zach Kelling

Lux Industries

2026-03-03

Abstract

We present Lux, a hybrid post-quantum consensus architecture for DAG-native chains. Lux separates fast operational finality from durable post-quantum finality. Validators emit Photons over a Nebula DAG substrate; classical BLS aggregates form Beams for low-latency confirmation, while ML-DSA attestations and Pulsar lattice-threshold Pulses provide post-quantum finality over Nebula roots. Prism binds all certificate lanes to the same transcript, and Quasar reaches Horizon finality when the bound lanes verify.

This document. Canonical wire formats for every Lux MPTC artifact: Pulsar shares, Pulsar signatures, Pulsar transcripts, Lens shares, Lens signatures, LSS snapshots, Horizon certificates, and Warp 2.0 envelopes. Where the wire format depends on length-prefix discipline, the discipline is normative and specified at byte resolution. The single source of truth is the canonical Go (`github.com/luxfi/pulsar`, etc.); this document specifies the format such that a C++ / WGSL / CUDA implementer can produce byte-equal output without reading the Go.

1 Encoding primitives

1.1 Length-prefixed bytes

A length-prefixed byte slice is encoded as:

```
[uint64 LE length] [length    byte payload]
```

Lengths are 8-byte little-endian unsigned integers. The Pulsar wire format uses this primitive throughout; `appendLenPrefixed` in `warp/pulsar/pulsar.go` is the canonical encoder.

1.2 TupleHash256 framing

TupleHash256 (NIST SP 800-185 [2]) takes a tuple of byte strings $(\tau_1, \ldots, \tau_k)$ and a personalization string S. It produces an unambiguous hash by encoding the lengths and the personalization into the absorption.

The Lux usage:

```
H_T(S, tau_1, ..., tau_k) := TupleHash256(input=(tau_1, ..., tau_k),
                                          customization=S,
                                          output_length=32 bytes)
```

A Pulsar transcript hash uses $S =$ "PULSAR-TRANSCRIPT-v1" (Section 4); a Quasar transcript uses $S =$ "QUASAR-TRANSCRIPT-v1"; a Warp 2.0 envelope uses $S =$ "WARP-PULSAR-ENVELOPE-v

2 Pulsar share

A Pulsar share is one validator's piece of the threshold key. Wire format:

```
PulsarShare := struct {
    Index        uint64       // 1-based party index
    KeyEraID     uint64       // group-key lineage
    Generation   uint64       // LSS resharing version
    GroupKeyHash [32]byte     // hash of (A, b_tilde) public key bytes
    ShareBytes   []byte       // s_i in NTT-Mont form, len-prefixed
    Seeds        [][]byte     // pairwise PRF seeds, len-prefixed each
    MACKeys      [][]byte     // pairwise MAC keys, len-prefixed each
    HashSuiteID  string       // e.g., "Pulsar-SHA3"
    Lambda       []byte       // Lagrange coefficient for this share, len-prefixed
}
```

The `ShareBytes` encoding mirrors Lattigo `v7`'s `Vector[Poly].WriteTo`: outer 8-byte vector length, then per-element 8-byte poly-levels-count followed by per-level (8-byte coeff-count + coeff-count × 8-byte uint64). See Section 3.1.

3 Pulsar signature

A Pulsar threshold signature is the output of Pulsar.Combine on K partial signatures. Wire format:

```
PulsarSignature := struct {
    C       Poly      // challenge polynomial, len-prefixed Lattigo Poly
    Z       Vector    // response vector, len-prefixed Lattigo Vector[Poly]
    Delta   Vector    // hint correction, len-prefixed Lattigo Vector[Poly]
}
```

Encoded byte stream:

```
[len(C bytes)]   [C bytes]
[len(Z bytes)]   [Z bytes]
[len(Delta bytes)] [Delta bytes]
```

A signature is ~ 33 KB on Pulsar mainnet parameters. `warp/pulsar/SerializePulse` produces this format; `warp/pulsar/DeserializePulse` parses it with the four-layer guards (Section 3.1).

3.1 Lattigo Poly / Vector wire format (normative)

A Lattigo v7 `Poly` is encoded as:

```
Poly := [
    uint64 LE      levels        // matrix outer length (typically 1 for ring-q)
    per level:
        uint64 LE   coeff_count
        coeff_count    uint64 LE  // the coefficient slice
]
```

A `Vector[Poly]` is encoded as:

```
Vector[Poly] := [
    uint64 LE      n              // number of polys in the vector
    n    Poly                       // each poly encoded as above
]
```

Hardening discipline. Every length-prefix is bounded by MaxLatticeUintSliceLen $= 4096$. Every nested slice's length is checked against the available bytes *before* lattigo's `ReadFrom` is invoked. The frame walkers `validatePolyFrame` and `validateVectorPolyFrame` (`warp/pulsar/pulsar.go`) implement this discipline; a malformed input is rejected before lattigo can recurse.

4 Pulsar transcript inputs

A Pulsar transcript hash is computed over:

```
TranscriptInputs := struct {
    ChainID, NetworkID, GroupID                    [32]byte
    KeyEraID, OldGeneration, NewGeneration         uint64
    OldEpochID, NewEpochID                         uint64
    OldSetHash, NewSetHash                         [32]byte
    ThresholdOld, ThresholdNew                     uint64
    GroupPublicKeyHash                             [32]byte
    NebulaRoot                                     [32]byte
    HashSuiteID                                    string
    ImplementationVersion                          string
    Variant                                        string  // "refresh" or "reshare"
    PrimaryNetworkID                               []byte  // empty for L1; set for L2
    PrimaryNebulaRoot                              [32]byte // zero for L1; bound for L2
}
```

Each field is length-prefix-encoded as part of the TupleHash256 input tuple. The order is canonical (per `pulsar/reshare/transcript.go`); two distinct field tuples yield distinct hashes by collision resistance of TupleHash256 with the `PULSAR-TRANSCRIPT-v1` personalization.

5 Pulsar activation message

```
ActivationMsg := "QUASAR-PULSAR-ACTIVATE-v1"
              || TranscriptHash(T)
              || ReshareTranscript.Hash(R)
```

Where $\mathcal{T}$ is the TranscriptInputs above and $\mathcal{R}$ encodes (commit digests, complaint hashes, disqualified senders, qualified quorum) per `pulsar/reshare/activation.go`.

6 Lens share and signature

Lens uses curve-side serialization:

```
LensShare := struct {
    Index        uint64
    KeyEraID, Generation uint64
    GroupKeyHash [32]byte
    ShareScalar  [32]byte    // big-endian scalar for Ed25519 / Ristretto255 / secp256k1
    HashSuiteID  string
    Lambda       [32]byte    // Lagrange coefficient
}

LensSignature := struct {
    R    [32]byte    // commitment point (compressed)
    s    [32]byte    // response scalar
}
```

Signature size is 64 bytes, regardless of group choice (Ed25519, secp256k1, Ristretto255). Curve-specific encoding follows the standard for each group.

7 LSS snapshot

```
LSSSnapshot := struct {
    Eta              uint64              // KeyEraID
    Generation       uint64
    RollbackFrom     uint64
    ValidatorSet     [][20]byte          // node IDs
    Threshold        uint64
    State            []byte              // encrypted share state, len-prefixed
    Timestamp        int64
    GroupKeyHash     [32]byte
    HashSuiteID      string
}
```

The `State` field is encrypted under a chain-configured snapshot key (typically a chain-internal KMS reference). Decryption is gated by snapshot-manager authority; the on-disk snapshot does not reveal share material to disk-level adversaries.

8 Horizon certificate

```
HorizonCertificate := struct {
    Beam         bls.AggregateCertificate    // 96-byte BLS aggregate
    MLDSA        mldsa.CertificateSet         // signer bitmap + per-validator sigs
                                              //   (or Z-Chain Groth16 rollup)
    Pulsar       pulsar.Certificate           // one Pulse, ~33 KB
    NebulaRoot   [32]byte
    KeyEraID     uint64
    GroupID      [32]byte
    Generation   uint64
    HashSuiteID  string
    Epoch        uint64
    Round        uint64
```

```
    ValidatorSetHash [32]byte
}
```

Each component is length-prefix-encoded. The wire format is the same in the Quasar consensus path and in the Warp 2.0 destination's lift-into-HorizonCertificate path; one canonical shape, two embedding contexts.

9 Warp 2.0 envelope

The canonical wire format:

```
0x02                                        // version byte (1 byte)
||
RLP([                                       // RLP-encoded list of 7 fields
    Message,                                // v1 Beam (UnsignedMessage +
        BitSetSignature)
    SourceNebulaRoot     [32]byte,          // canonical bundle root
    SourceKeyEraID       uint64,            // Pulsar lineage
    SourceGeneration     uint64,            // LSS lineage
    HashSuiteID          string,            // e.g. "Pulsar-SHA3"; "" defaults
    PulsarPulse          []byte,            // empty if absent
    MLDSACertSet         []byte             // empty if absent
])
```

Why RLP and not protobuf. Warp 1.x already uses RLP (Ethereum recursive length prefix [1]) for compatibility with EVM clients. Warp 2.0 maintains the RLP discipline so that v1 receivers see v2 bytes as "not a valid RLP-list prefix" and reject. Switching to protobuf or any other encoding would break the forward-only migration discipline.

Wire bounds. `MaxEnvelopeV2Size = 4 × MaxMessageSize`, conservatively bounded for the optional Pulse and ML-DSA cert-set bytes.

10 Cross-implementation byte-equality

The Go canonical produces byte-equal output across runs for every artifact in this spec. KAT vectors gate every release:

- `pulsar/cmd/ringtail_oracle_v2/`: emits 16 KAT entries covering every Pulsar wire object.
- `pulsar/scripts/regen-kats.sh`: deterministic regeneration; `-verify` confirms byte-equality.
- `warp/pulsar/testdata/`: representative Warp 2.0 envelope KAT.

A C++ / WGSL / CUDA implementer SHOULD produce byte-equal output against these KATs. Where this spec and the canonical disagree, the canonical is the tiebreaker; a discrepancy is a spec bug, not an implementation bug.

11 Endianness, alignment, padding

- Endianness: little-endian for all uint64 fields (consistent with Lattigo v7's wire format).
- Alignment: no implicit alignment; lengths and payloads are byte-packed.
- Padding: no padding. A trailing-byte difference between two encodings of the same logical content is a wire-format error.

12 Versioning

Every wire format that may evolve carries an explicit version byte or version field:

- Warp envelope: `0x01` (legacy v1, no version byte) and `0x02` (v2). No v3 reserved.
- Pulsar transcript: `ImplementationVersion` string field (e.g., `"pulsar-v1.0"`).
- Hash suite: `HashSuiteID` string field (e.g., `"Pulsar-SHA3"`).

A version mismatch returns an explicit error; there is no negotiation, no fallback, no implicit upgrade.

References

[1] Ethereum Foundation. Recursive length prefix (RLP). https://github.com/ethereum/wiki/wiki/RLP, 2014.

[2] NIST. Nist sp 800-185: Sha-3 derived functions. https://csrc.nist.gov/pubs/sp/800/185/final, 2016.