Warp 2.0: Pulse-Backed Cross-Chain Source Finality

Zach Kelling

Lux Industries

2026-03-03 *Subtitle: An envelope evolution carrying source-chain Horizon finality across chain boundaries via Pulsar Pulses.*

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 paper. Warp 2.0 is the cross-chain envelope that carries source-chain Horizon finality across chain boundaries. Warp 1.x [13] signs cross-chain messages with a BLS aggregate over the source chain's validator-key registry; that classical-only design gives a destination chain no PQ guarantee about the source's state. Warp 2.0 adds a Pulsar Pulse over a canonical transcript binding the source chain's NebulaRoot, KeyEraID, and Generation. The destination verifier runs Prism over the source's transcript using the source-chain key registry. Beam, ML-DSA, and Pulse all lift their soundness arguments from the consensus side (Quasar Horizon [7, 10]) to the cross-chain side via transcript binding, with one explicit theorem (Warp 2.0 envelope unforgeability, `proofs/quasar/warp-pq-soundness.tex`) covering the cross-chain case. We document the wire format, the v1 → v2 migration discipline (forward-only; v1 verifiers reject the leading `0x02` byte), the L2 transitive inheritance, and the constant-time fuzz harness that surfaced one vulnerability (a lattigo ReadUint64Slice DoS) which is fixed in production. Warp Private (FHE-encrypted payload, optionally Pulse-signed) is orthogonal and discussed only briefly here.

Contents

1 Introduction

Lux is not merely adding post-quantum signatures to a chain; it defines a hybrid finality architecture for DAG-native consensus, with protocol-agnostic threshold lifecycle, post-quantum threshold sealing, and cross-chain propagation of Horizon finality.

This paper specifies the cross-chain leg of that thesis. Warp 2.0 evolves the Lux cross-chain messaging protocol (LP-021 [11], `github.com/luxfi/warp` [15]) to carry source-chain Horizon finality across chain boundaries via a Pulsar Pulse over a canonical envelope transcript. The receiving chain's verifier runs Prism over the source chain's transcript, producing a destination-side Horizon-final acceptance whose PQ root of trust resolves to the source's Pulsar GroupKey, not to a bridge or an oracle.

1.1 Why Warp 2.0

Warp 1.x is shipping (LP-021, LP-075 [11, 13]). It signs cross-chain messages with a BLS aggregate over the source chain's validator-key registry. The classical-only design has three structural shortcomings:

1. **No post-quantum guarantee.** A quantum adversary that breaks BLS in polynomial time can forge a Warp 1.x envelope from any source chain to any destination, with no fallback PQ check. The destination has no cryptographic basis to discriminate honest from forged Warp 1.x bytes.
2. **No source-finality binding.** A Warp 1.x envelope binds a source chain id, a payload, and a $\geq 2/3$-weight BLS aggregate. It does not bind *when* on the source the message was finalized: the same source-chain validator set could sign multiple conflicting payloads, and Warp 1.x has no transcript field that anchors the message to a specific source-chain NebulaRoot.
3. **No source-key-era lineage.** A Warp 1.x envelope does not carry the source's KeyEraID / Generation tuple. A destination that wants to verify against the source's current key state has to reconcile the envelope's BLS aggregate against a registry view that may have shifted post-signing.

Warp 2.0 fixes all three. The shape is direct: take the source-chain Horizon certificate (Beam + ML-DSA + Pulse, all bound to the source's NebulaRoot via the canonical `QuasarTranscript`); embed it in a cross-chain envelope; bind the envelope payload to the same transcript via a canonical `WARP-PULSAR-ENVELOPE-v1` prefix; have the destination verifier run Prism over the source-chain transcript using the source-chain key registry. The result: a cross-chain envelope that, if Prism-accepted, is post-quantum-sound under the same composition theorem (Theorem 1, Section 5) as its consensus cousin.

1.2 Forward-only migration

Warp 1.x and Warp 2.0 share the wire space. The discipline is forward-only:

- A Warp 2.0 envelope begins with a leading `0x02` byte. Warp 1.x verifiers see this byte first, fail to parse it as a valid RLP-list prefix, and reject. This is the correct refusal: a v1 verifier cannot validate v2 transcript binding, so it must not act on v2 bytes.
- A Warp 2.0 receiver decodes Warp 1.x bytes via the same dispatcher (`ParseEnvelope`) that handles the v2 path. Warp 1.x bytes have no leading version byte; the dispatcher detects v2's leading byte and falls back to v1 parsing otherwise. The result is a v2 envelope with only the Beam lane populated and all v2 fields zero-valued.
- Senders that need to support v1-only verifiers emit Warp 1.x bytes on the v1 channel. The same source-chain UnsignedMessage may be embedded in a v2 envelope on the v2 channel without re-signing the Beam.

There is no v1 ↔ v2 negotiation. There is no protocol downgrade. There is no v3 reserved that "maybe re-encrypts the wire." The only forward step from Warp 2.0 is Warp Private (FHE-encrypted payload, optionally Pulse-signed), which is orthogonal to v2 (Section 2.4).

1.3 What this paper specifies

Section 2 traces the Warp version history (1.x → 1.5 hybrid → 2.0 Pulse-backed → Warp Private). Section 3 gives the canonical wire format and cites the transcript-binding definition. Section 4 specifies the cross-chain binding fields (sourceNebulaRoot, sourceKeyEraID, sourceGeneration) and the L1/L2/L3 transitive inheritance. Section 5 states the cross-chain soundness theorem and its connection to Quasar Horizon. Section 6 documents the lattigo ReadUint64Slice DoS that the fuzz harness surfaced and that is fixed in production. Section 7 cites the running Go canonical (`warp/envelope.go`, `warp/pulsar/pulsar.go`) and the v1↔v2 compatibility tests. Section 8 compares to CCIP, Wormhole, IBC, and LayerZero, with emphasis on the Pulse-backed PQ source finality angle.

1.4 What this paper does not do

We do not specify the source-chain Horizon construction (that is [7]). We do not specify Pulsar (that is LP-073 [12, 6]) or LSS (that is [18, 14]). Where the underlying threshold mathematics is needed, we cite the canonical proof-bucket files (`proofs/quasar/warp-pq-soundness.tex`, `proofs/pulsar/unforgeability.tex`, `proofs/definitions/transcript-binding.tex`). We do not duplicate proof bodies; we cite them.

2 Warp Evolution

Lux Warp evolves on the same lane structure as Quasar finality. This section traces the version history.

2.1 Warp 1.x: BLS aggregate

LP-021 / LP-075 [11, 13]. Source-chain validators emit a BitSetSignature (signer bitmap plus aggregated BLS signature, 96 bytes total) over an UnsignedMessage that binds the source networkID, sourceChainID, and a payload. The destination chain verifies the aggregate against the source's registered validator set (BLS public keys, weights). Warp 1.x is shipping; the corresponding code lives at `github.com/luxfi/warp/message.go`, `signature.go`, and `validator.go` [15].

What 1.x gives. Classical fast cross-chain. Sub-millisecond destination-side BLS aggregate verify (Section 7). Operationally simple: one signing primitive, one verifier.

What 1.x does not give. (1) Post-quantum guarantee on the cross-chain envelope: a quantum break of BLS forges any cross-chain message. (2) Source-finality binding: no transcript field anchors the envelope to a specific source-chain bundle root. (3) Source-key-era lineage: the validator set the destination resolves at verify time may differ from the validator set the source signed with.

2.2 Warp 1.5: hybrid (transitional)

Warp 1.5 was a transitional design that added an optional ML-DSA cert set alongside the BLS aggregate. It bound to the source chain id and payload but not to a Nebula root or a key-era lineage. In production it added per-validator PQ accountability without the compact PQ threshold seal.

Status. Superseded by Warp 2.0. Code never shipped to mainnet; the design persisted only long enough to validate that the Pulse field could compose with the existing Message shape without breaking the Warp 1.x BLS path. The Mar-3 freeze ships Warp 2.0; Warp 1.5 is documented here for historical completeness.

2.3 Warp 2.0: Pulse-backed cross-chain finality

The current target. A Warp 2.0 envelope carries:

- the v1 Message unchanged — so the BLS Beam path remains bit-equal;
- four transcript-binding fields (sourceNebulaRoot, sourceKeyEraID, sourceGeneration, HashSuiteID);
- one Pulsar Pulse over the canonical `WARP-PULSAR-ENVELOPE-v1` transcript;
- optionally an ML-DSA cert set (per-validator FIPS-204 attestations) over the same transcript.

The destination verifier (`warp/pulsar/KernelVerifier` [15]) resolves (sourceChainID, sourceKeyEraID, source to the source's $\mathsf{GroupKey}_\eta$ via the destination chain's source-chain key registry; runs Pulsar.Verify on the Pulse; runs the BLS aggregate verify on the v1 Beam; runs the ML-DSA cert-set verify if the lane is present; returns Prism-accepted iff all lanes verify against the same transcript.

2.4 Warp Private: FHE-encrypted payload

Z-Chain FHE ciphertext envelope. The Pulsar pulse signs the ciphertext digest; the destination verifies under unchanged $\mathsf{GroupKey}_\eta$ without seeing plaintext. Warp Private is *production-research*, orthogonal to Warp 2.0; the two compose: a Warp Private envelope MAY carry a Pulse, in which case the v2.0 soundness theorem (Theorem 1) applies unchanged to the encrypted-payload binding. See Remark 2 of `proofs/quasar/warp-pq-soundness.tex`.

2.5 Future PQ proof system

The current Z-Chain rollup uses Groth16 for ML-DSA cert-set compression; a future Warp Private PQ replaces the pairing-based SNARK with P3Q [3] or a lattice-based transparent proof system. Until that lands, do not classify the Groth16-wrapped lane as post-quantum (Remark 1, Section 5).

2.6 Version table

Version	Signing lane(s)	Status	Description
Warp 1.x	Beam (BLS aggregate)	Shipping	Classical cross-chain. No PQ guarantee.
Warp 1.5	Beam + optional ML-DSA	Superseded	Hybrid transitional; never deployed.
Warp 2.0	Beam + ML-DSA + Pulse	Shipping (Mar-3 freeze)	Pulse-backed source finality. PQ-sound under the conjunctive Prism predicate.
Warp Private	Pulse over FHE ciphertext digest	Production-research	Confidentiality + PQ source finality, orthogonal composition.
Warp Private PQ	Pulse + P3Q [3] / lattice ZK wrapper	Future	Replaces Groth16 wrapper for true PQ succinctness.

The Mar-3 freeze ships Warp 2.0 in production. Warp 1.x continues to ship for backwards compatibility; the v1 channel and the v2 channel coexist without negotiation, per the forward-only migration discipline (Section 1).

3 Envelope Format

This section gives the canonical wire format for the Warp 2.0 envelope. The single source of truth for transcript binding is `proofs/definitions/transcript-binding.tex`; the canonical Go source is `github.com/luxfi/warp/envelope.go` [15]. Where this paper and the Go source disagree, the Go source is the tiebreaker.

3.1 Wire layout

A serialized Warp 2.0 envelope has the form:

```
0x02 || RLP([
    Message,                        // v1 Beam (UnsignedMessage + BitSetSignature)
    SourceNebulaRoot    [32]byte,   // canonical bundle root of source chain
    SourceKeyEraID      uint64,     // source chain Pulsar lineage
    SourceGeneration    uint64,     // source chain LSS resharing version
    HashSuiteID         string,     // e.g. "Pulsar-SHA3"; "" defaults
    PulsarPulse         []byte,     // Pulsar threshold sig; empty if absent
    MLDSACertSet        []byte,     // ML-DSA cert set; empty if absent
])
```

Leading version byte. `EnvelopeVersion2 = 0x02`. The byte is unconditionally present at offset 0 of every Warp 2.0 envelope. A Warp 1.x receiver attempting to parse a v2 envelope as a v1 message rejects on the first byte (RLP lists begin with `0xc0`–`0xff`; the bare `0x02` is not a valid RLP-list prefix). This is the correct refusal: a v1 verifier cannot validate v2 transcript binding ([15] `envelope.go` L43–L65 in the canonical commit).

Field count is fixed. All seven fields after the version byte are unconditionally present in the RLP list. Optional PQ lanes are signaled by zero-length byte slices when absent. This is deliberate: the field count is fixed, decoders only need to inspect lengths, and a future v2 extension cannot ambiguously truncate the list.

Maximum size. `MaxEnvelopeV2Size = 4 * MaxMessageSize`, conservatively bounded to leave room for the optional Pulse and ML-DSA cert-set bytes alongside the v1 message ([15] `envelope.go` L37).

3.2 Canonical transcript

The Pulse signs H_T over the canonical `WARP-PULSAR-ENVELOPE-v1` transcript:

```
WarpEnvelopeV2Transcript := H_T(
    "WARP-PULSAR-ENVELOPE-v1",
    sourceChainID,            // 32-byte source chain id
    payload,                  // the v1 Message's UnsignedMessage payload
    sourceNebulaRoot,         // 32-byte canonical bundle root
    sourceKeyEraID,           // uint64
    sourceGeneration,         // uint64
)
```

This is Definition **??** of `proofs/definitions/transcript-binding.tex`. The personalization tag is canonical and *not* re-used:

- `QUASAR-PULSAR-BUNDLE-v1` signs Nebula bundle roots in consensus.

- `QUASAR-PULSAR-ACTIVATE-v1` signs Pulsar key-era activation messages.
- `WARP-PULSAR-ENVELOPE-v1` signs cross-chain message envelopes.

A Pulse over a Nebula bundle root cannot be replayed as a Pulse over a Warp envelope, and vice versa. Re-using one prefix for both would let a consensus-side Pulse forge a cross-chain envelope (or vice versa); this is explicitly rejected by the domain-separation discipline.

3.3 Why `HashSuiteID` is bound

Every Pulsar deployment pins a hash suite identifier (`Pulsar-SHA3` as the production default, `Pulsar-BLAKE3` as an optional non-normative profile). The suite identifier is bound into every transcript via TupleHash personalization (`proofs/definitions/transcript-binding.tex` Definition **??**). Cross-suite signatures do not collide; a forger cannot leverage a signature under one suite to forge under another (`proofs/pulsar/hash-suite-separation.tex` Theorem **??**).

The Warp 2.0 envelope's HashSuiteID field commits the verifier to the source chain's suite at signing time. If the destination's resolver returns a HashSuiteID that does not match the envelope's, the verifier returns `ErrSuiteMismatch` ([15] `warp/pulsar/pulsar.go` L78–80). This catches the case where the source chain has Reanchored to a new suite but the destination has not yet seen the Reanchor: rather than verifying under a wrong suite, the destination correctly refuses.

3.4 Backward compatibility

A Warp 1.x receiver is presented with raw v2 wire bytes. The first byte `0x02` is not a valid RLP-list prefix (RLP lists begin `0xc0`–`0xff`); the v1 decoder rejects. This is a hard refusal, not a downgrade.

A sender that needs to reach v1-only verifiers MUST emit Warp 1.x wire bytes (call `Message.Bytes()`) on the v1 channel. The same UnsignedMessage may be embedded in a v2 envelope on the v2 channel without re-signing the Beam; the BLS aggregate over UnsignedMessage.Hash() is bit-equal in both channels.

3.5 Forward compatibility

A Warp 2.0 receiver decoding Warp 1.x bytes uses `ParseEnvelope`, which detects the leading byte. If the first byte is not `0x02`, the dispatcher falls back to `ParseMessage` and lifts the result into a v2 envelope with PQ lanes empty:

```
EnvelopeV2 {
    Message:          v1 Message,        // populated
    SourceNebulaRoot: [32]byte{0...},    // zero
    SourceKeyEraID:   0,                 // zero
    SourceGeneration: 0,                 // zero
    HashSuiteID:      "",                // empty
    PulsarPulse:      nil,               // absent
    MLDSACertSet:     nil,               // absent
}
```

The v2 verifier sees the empty PQ lanes and falls back to the Beam-only verification path. This is the correct semantics: a v1 envelope is observable on the v2 channel as a Beam-final-only confirmation, and the destination's downstream consumers can choose whether to act on Beam-only or to require Pulse-backed.

3.6 No v3 reserved

There is no v3 reserved byte. The version space is small by design (1 = legacy, 2 = current). Future extensions (Warp Private, Warp Private PQ) re-use the v2 envelope shape with additional optional fields, gated by the chain's governance — not by a new wire version. This rejects the "every protocol bump is a wire bump" anti-pattern; the wire is stable, the policy on top of it evolves.

3.7 Forward-only migration discipline

There is no v1 ↔ v2 negotiation. There is no protocol downgrade. There is no shared-protocol fallback that lets a v2 sender be "polite" to a v1-only receiver: senders that need to reach v1-only receivers emit v1 bytes on the v1 channel. The two channels coexist; the wire bytes do not negotiate.

This is the LP-105 [16] deprecation timeline made wire-explicit: "no backwards compatibility only forwards perfection." The version space is for honest version identification, not for protocol-level fallback.

4 Cross-Chain Binding

A Warp 2.0 envelope carries three lineage scalars from the source chain. This section specifies what each one binds and how the destination verifier uses it.

4.1 The three lineage scalars

- sourceNebulaRoot: the source chain's canonical bundle root for the bundle that the Pulse signs. A 32-byte digest binding the source chain's validator-set hash, parent epoch, and bundled vertices (Section **??** of [7]; equivalently `proofs/definitions/transcript-binding.tex` Definition **??**).
- sourceKeyEraID: the source chain's Pulsar group-key lineage ID at the time of signing. Bumps only at Reanchor. Stable across LSS Reshare / Refresh within the era.
- sourceGeneration: the source chain's LSS resharing generation ID at the time of signing. Bumps every Refresh / Reshare under the same GroupKey.

The destination chain resolves (sourceChainID, sourceKeyEraID, sourceGeneration) to the source's GroupKey + HashSuiteID via a GroupKeyResolver ([15] `warp/pulsar/pulsar.go` L82–97).

4.2 The GroupKey resolver

```
type GroupKeyResolver interface {
    ResolveGroupKey(
        sourceChainID [32]byte,
        keyEraID uint64,
        generation uint64,
    ) (gk *pulsarKernel.GroupKey, suiteID string, err error)
}
```

The resolver is implemented by the destination chain against its source-chain key registry — a contract that records the source's GroupKey lineage as it evolves through Bootstrap, Reshare, and Reanchor events. Returning a zero-pointer GroupKey or empty suiteID is treated as ErrGroupKeyResolverFailed.

Within an era. For every g within the same η, the GroupKey is byte-identical (Lemma **??** of `proofs/lss/lifecycle-safety.tex`). Therefore the resolver only needs to learn the source's η on Reanchor; intermediate Refresh / Reshare events do not require new resolver state.

Across eras. A Reanchor event on the source chain produces a new η and a new GroupKey. The destination's resolver must observe the Reanchor (typically by validating a destination-side update transaction that carries the Reanchor's activation cert) before it can verify v2 envelopes signed under the new η.

Why this is not a centralized bridge oracle. The resolver does not introduce trust beyond the destination chain itself: the resolver's state is on-chain on the destination, updated via destination-side transactions whose finality is itself Horizon-final on the destination. The trust model is destination Horizon + source Horizon (the two compose); there is no third-party oracle.

4.3 Transitive L1 / L2 / L3 inheritance

The L1 / L2 / L3 tier model (Section **??** of [7], Definition **??**) extends to Warp 2.0 cross-chain envelopes through transcript binding:

Source = L1 (sovereign). The source chain's Horizon-final binds its own validator-set Pulse + ML-DSA. The Warp 2.0 envelope's Pulse inherits from the source's GroupKey at (η, g). The destination's verifier returns Horizon-final acceptance iff Prism accepts.

Source = L2 (co-validating with primary N_P). The source chain's activation transcript binds primary_nebula_root from N_P. The Warp 2.0 envelope's Pulse over the source's Nebula-Root transitively inherits from N_P's validator set: a destination receiving an L2 Warp message verifies the L2's Pulse and, transitively, the L2's anchor to its primary network — without trusting the L2 in isolation. The transitive inheritance is Theorem **??** of `proofs/quasar/l1-l2-covalidation.tex`, applied at the cross-chain layer.

Source = L3 (rollup / validity / app-execution). The source's Horizon-final inherits from the underlying L1/L2's Pulse + ML-DSA via the L3's chosen proof system (Groth16, STARK, FHE circuit). The Warp 2.0 envelope from an L3 carries the L3's Pulse over the L3's bundle root; the destination's resolver maps the L3's sourceKeyEraID to the underlying L1/L2's GroupKey at the bound moment. The L3's proof system is a compatibility / compression / privacy adapter, not the PQ root of trust (Remark 1, Section 5).

4.4 Cross-chain validator-set evolution

A common operational worry: "what if the source chain rotates its validator set between the time the Warp envelope is signed and the time it lands on the destination?" Warp 2.0's answer: bind the (η, g) at signing, resolve the GroupKey accordingly. Validator rotation within an era does not change the GroupKey (Lemma **??**), so the destination resolver can verify against the stable $\mathsf{GroupKey}_\eta$ without reconciling the precise g. Across-era rotation (Reanchor) does change the GroupKey, and the destination's resolver must observe the Reanchor before envelopes signed under the new η verify.

The discipline is symmetric: a source-chain Reanchor produces a new η; envelopes signed under the new η are unverifiable on the destination until the destination's resolver state has been updated; updating the resolver state is a destination-side transaction whose own finality is Horizon-final. There is no out-of-band bridge component.

4.5 Cross-chain Horizon batching

Multiple source NebulaRoots may be packed into a single Pulse with a Merkle root over the batch. The transcript becomes:

```
WarpEnvelopeV2BatchedTranscript := H_T(
    "WARP-PULSAR-ENVELOPE-BATCHED-v1",
    sourceChainID,
    Merkle(payload_1, payload_2, ..., payload_k),
    Merkle(sourceNebulaRoot_1, ..., sourceNebulaRoot_k),
    sourceKeyEraID,
    sourceGeneration,
)
```

The destination provides Merkle proofs to verify individual payloads against the batched transcript. This is an extension; the Mar-3 freeze ships single-payload v2 envelopes.

4.6 Replay protection

A Warp 2.0 envelope is replay-safe across chain boundaries: the sourceChainID binds the originator, and the destination's chain id is bound via the destination's payload-level conventions (e.g., `AddressedCall` carries a destination chain id). At the source-finality level, the sourceNebulaRoot pins the bundle: replaying the same envelope at a later source-bundle would not match the destination's resolver because the resolver verifies the envelope's (sourceKeyEraID, sourceGeneration) matches what the source had at the bundle.

The PQ lanes are not vulnerable to replay across protocols: the personalization tag `WARP-PULSAR-ENVELOPE-v` differs from the consensus-side tag `QUASAR-PULSAR-BUNDLE-v1`, so a Pulse over a bundle root cannot be lifted into an envelope (Section 3).

5 Soundness

The cross-chain soundness theorem is `proofs/quasar/warp-pq-soundness.tex`. We restate it and connect it to the per-lane reductions and to the consensus-side Horizon theorem.

5.1 Warp 2.0 envelope unforgeability

Theorem 1 (Warp 2.0 envelope unforgeability)**.** *Let* env_2 *be a Warp 2.0 envelope (Definition* **??***) with successful verification under* Verify_{V2}*. Then forging an alternative* env_2' *on the same* (sourceChainID, sourceNebulaRoot, sourceKeyEraID, sourceGeneration) *tuple with a different* payload *requires breaking either:*

1. *BLS12-381 co-CDH (forging the Beam component); OR*
2. *Module-LWE / Module-SIS (forging the Pulsar Pulse, Theorem* **??***).*

A quantum adversary breaking only co-CDH (and not MLWE/MSIS) succeeds against the Beam alone; the Pulse remains unforgeable, so the v2 envelope as a whole remains sound under Verify_{V2}*.*

The full proof is Theorem 1 of `proofs/quasar/warp-pq-soundness.tex`; the per-lane reductions live in `proofs/pulsar/unforgeability.tex` (Pulsar SUF-CMA) and are companion-cited from `quasar-cert-soundness.tex` [8]. We highlight three structural facts:

The conjunction is essential. The shape "Verify_{V2} accepts iff Beam-OR-Pulse" is wrong for the same reason that "Horizon iff Beam-OR-Pulse" is wrong (Section **??** of [7]; [5]): under a quantum adversary that breaks only BLS, "Beam-OR-Pulse" degrades to "Beam alone" and the envelope becomes forgeable. Warp 2.0's Verify_{V2} is conjunctive: every required lane must verify against the same canonical `WARP-PULSAR-ENVELOPE-v1` transcript.

PQ finality lives in the Pulse. The Pulse is the post-quantum root of trust for a Warp 2.0 envelope. The Beam provides classical fast-path verification (sub-millisecond on commodity hardware). The optional ML-DSA cert set provides per-validator PQ accountability and slashing-evidence material. Under a quantum break of BLS, the Beam falls; the envelope remains sound iff the Pulse verifies. Without the Pulse, the v2 envelope degrades to Warp 1.x semantics — which is correct, because absent the Pulse, the envelope is exactly Warp 1.x.

The Groth16 wrapper is not part of the PQ reduction.

Remark 1 (Groth16 wrapper does not contribute PQ security to Warp 2.0)**.** *A Z-Chain Groth16 rollup proof of ML-DSA cert-set verification (Definition* **??***) is a* classical succinct proof of post-quantum signature verification. *The proof system is pairing-based and not post-quantum. Warp 2.0's PQ soundness (Theorem 1 parts 2 and the* Pulsar.Verify *branch) flows from the ML-DSA signatures and Pulsar threshold signature alone; the Groth16 proof is verified for compatibility / compression only and is not part of the PQ reduction. See Remark* **??** *of* `proofs/quasar/horizon-soundness.tex`*.*

The Mar-3 freeze ships Warp 2.0 with the Pulse as the PQ root of trust. A future Warp Private PQ envelope replaces Groth16 with P3Q [3] or a lattice-based wrapper; PQ finality continues to flow from ML-DSA + Pulsar.

5.2 Per-lane reductions

Beam (BLS aggregate). Forging the v1 Message (which is the Beam carrier inside the v2 envelope) requires either $> 2/3$-weight BLS-key control, or a forgery in the BLS12-381 signature scheme. Reduces to co-CDH [1]. Quantum-broken in polynomial time.

Pulse (Pulsar SUF-CMA). Forging a Pulse on the canonical `WARP-PULSAR-ENVELOPE-v1` transcript with at most $t-1$ corrupted parties reduces to MLWE / MSIS, by Theorem **??** of `proofs/pulsar/unforgeability.tex`. With LP-073 parameters [6], classical security $\geq 2^{142}$ and quantum security $\geq 2^{130}$.

ML-DSA cert set (FIPS 204 EUF-CMA). Each per-validator signature reduces to MLWE / MSIS by FIPS 204's standard analysis. The cert-set's $\geq 2/3$-weight requirement adds no structural shortcut beyond the per-validator forgery cost.

5.3 Cross-lane independence

Lemma 2 (Lane independence under quantum advice, cross-chain)**.** *Let $\mathcal{A}$ be a polynomial-time adversary with quantum oracle access to BLS12-381's discrete log. Then $\mathcal{A}$'s success probability against the Pulsar lane in a Warp 2.0 envelope is bounded by* $\mathrm{Adv}^{\mathsf{MLWE}}_{q,\sigma_E}(\mathcal{A}) + \mathrm{Adv}^{\mathsf{MSIS}}_{q,[\mathbf{A}|-c\tilde{\mathbf{b}}],2B}(\mathcal{A}) + \mathrm{negl}(\lambda)$.

This is the cross-chain analog of Lemma **??** of [7]: BLS provides no PQ contribution to a multi-lane cert, and the Pulse / ML-DSA reductions are independent of the BLS reduction.

5.4 Warp Private orthogonality

Remark 2 (Warp Private orthogonality)**.** Warp Private *(FHE-encrypted payload, optionally Pulse-signed) is orthogonal to* Warp 2.0*. Warp Private provides confidentiality; Warp 2.0 provides PQ finality. They compose: a Warp Private envelope MAY carry a Pulse, in which case Theorem 1 applies unchanged to the encrypted-payload binding. Until the FHE circuits are production-ready, the Mar-3 freeze ships only Warp 2.0; Warp Private is open work.*

5.5 Mechanization status

- **Warp 2.0 envelope unforgeability.** `proofs/quasar/warp-pq-soundness.tex`.
- **Pulsar SUF-CMA.** Lean: `Crypto.Pulsar.corona_pq_unforgeability`.
- **Reshare preserves master secret.** Lean: `Crypto.Threshold.Reshare.reshare_preserves_public_ke`
- **LSS lifecycle invariants.** `proofs/lss/lifecycle-safety.tex`, `proofs/lss/rollback-safety.tex`.
- **Tamarin.** `tamarin/QuasarConsensus.spthy` verifies the protocol-level handshake / vote / bundle flow that produces the source-side Horizon certificate that Warp 2.0 carries.

5.6 Why we do not assume synchrony for Warp delivery

Warp envelope delivery is asynchronous; the destination's verifier accepts an envelope when all required lanes verify against the same transcript, irrespective of inter-chain timing. There is no global clock assumption. A Warp 2.0 envelope sits in the destination's pending-envelopes queue until the destination's resolver has the source's (η, g) resolved; once resolved, the envelope is verifiable.

This is the cross-chain analog of asynchronous PQ finality (Section **??** of [7]). The forward consequence: cross-chain finality lag is bounded by source-chain Horizon-final lag plus destination-chain resolver-update lag, both of which are governance-configurable.

6 Fuzz-Discovered Vulnerability: lattigo `ReadUint64Slice` DoS

The Warp 2.0 implementation went through a constant-time review and a fuzz harness pass before the Mar-3 freeze. The fuzz harness (`warp/pulsar/fuzz_pulse_test.go`, `warp/pulsar/fuzz_horizon_cert` `warp/fuzz_envelope_test.go`) surfaced one production-relevant vulnerability in the `luxfi/lattice/v7` [9] deserializer that the Pulsar wire format depends on. This section documents the finding, the exploit shape, and the fix.

6.1 The bug

The lattigo v7 fork carries `utils/buffer/reader.go` `ReadUint64Slice`, which recursively reads a uint64 length-prefix followed by a uint64 slice of the declared length. The recursion happens via a partial-peek buffer call: the reader peeks at the length, allocates a slice, and recurses to fill it. If the declared length is unbounded by the available bytes, the recursion descends without making progress.

The exploit. An attacker constructs a wire-format Poly with an inner length-prefix of `0xFFFFFFFFFFFFFFFF` (or any large value) and only a few bytes of payload. The lattigo deserializer reads the length, allocates a slice of the declared size, and recurses to fill it. The recursion descends without resolving; the Go runtime stack overflows; the process dies.

Where it lands. On the destination chain. Specifically: a destination receiving a Warp 2.0 envelope with an attacker-crafted Pulse byte stream, calling `warp/pulsar.DeserializePulse`, which delegates to the lattigo deserializer. The destination chain's verifier process panics; the chain's verifier loop crashes. Repeated envelopes denial-of-service the destination's Warp message ingress.

Severity. High. A non-validator attacker can DoS arbitrary destination chains by sending malformed Warp 2.0 envelopes. The Beam aggregate inside the envelope is irrelevant to the attack; the lattigo deserializer is invoked before the envelope is structurally valid. The attack surface is the public Warp 2.0 message channel.

6.2 The fuzz harness that found it

The fuzz harness at `warp/pulsar/fuzz_pulse_test.go` ([15], file head):

```
// FuzzPulseDeserialize runs DeserializePulse over arbitrary bytes and
// confirms it never panics.
func FuzzPulseDeserialize(f *testing.F) {
    // Seed: a real pulse from a 3-of-2 ceremony.
    good := mustGoodPulseSeed()
    f.Add(good)
    f.Add([]byte{})
    f.Add([]byte{0x00})
    f.Add(bytes.Repeat([]byte{0xFF}, 1024))
    // ...
    f.Fuzz(func(t *testing.T, raw []byte) {
        _, err := DeserializePulse(raw)
        // Property: errors are fine; panics are not.
        // (The defer-recover in DeserializePulse turns lattigo
        // panics into clean errors.)
        _ = err
    })
}
```

The harness ran for 60 s under `go test -fuzz=FuzzPulseDeserialize` and quickly produced a corpus entry of $\sim$ 30 bytes that triggered the lattigo recursion. The diagnostic was a stack overflow in `lattigo/v7/utils/buffer/reader.go:ReadUint64Slice`; the stack trace showed unbounded recursion through the length-prefix code path.

6.3 The fix

The fix lives in `warp/pulsar/pulsar.go` ([15] L257–L327 in the canonical commit) in three layers:

Layer 1: outer wire-size cap.

```
const MaxPulseWireSize = 64 * 1024
```

A real Pulsar lattice threshold signature is $\sim$ 33 KB (LP-073 [12] §"Wire Format"); the cap is 64 KB to leave headroom for ring-parameter changes. The cap doubles as a recursion bound: lattigo's ReadUint64Slice recurses by length field, so bounding the byte stream bounds the recursion depth (64 KB $\div$ 8 = 8192 frames worst case, well under Go's default 8 MB goroutine stack at 64 B per frame).

Layer 2: per-lane frame caps.

```
const MaxPulseFrameSize = 32 * 1024
const MaxLatticeUintSliceLen = 4096
```

MaxPulseFrameSize bounds each individual lane (C, Z, Δ) inside a serialized pulse. MaxLatticeUintSliceLen bounds the largest uint64 slice inside any lattigo wire frame. A canonical Pulsar Poly has 256 coefficients per level; a Vector / Matrix has at most $M \cdot N = 8 \cdot 32 = 256$ polys; every inner length-prefix should fit comfortably under 4096. A frame whose declared inner length exceeds this is a structural red flag and is rejected before lattigo's ReadFrom can recurse.

Layer 3: end-to-end frame walker. `validatePolyFrame` and `validateVectorPolyFrame` parse the Poly / Vector[Poly] wire format byte-by-byte before calling lattigo. Every length-prefix is checked against MaxLatticeUintSliceLen; every nested coefficient slice is checked against the available bytes. If the frame is malformed (too short, length-prefix exceeds bound, trailing bytes), the validator returns a clean error and the lattigo deserializer is never invoked.

Layer 4: defer-recover boundary. The `DeserializePulse` entry point installs a `defer recover()` handler that converts any lattigo panic that escapes the validators into a clean error ([15] `warp/pulsar/pulsar.go` L389–L398):

```
defer func() {
    if r := recover(); r != nil {
        sig = nil
        err = fmt.Errorf("warp pulsar: deserialize panic recovered: %v", r)
    }
}()
```

This is defense in depth: the frame walker should catch all malformed inputs, but if a future lattigo change introduces a new panic path, the recover boundary ensures the destination chain's verifier process does not crash.

6.4 Why all four layers

The combination is belt-and-suspenders by design:

- Layer 1 (wire-size cap) is the cheapest check; rejects 99% of attack inputs in O(1).
- Layer 2 (per-lane caps) catches inputs that fit under the wire cap but blow up an inner frame.
- Layer 3 (frame walker) catches inputs whose lengths add up but whose internal structure recurses unboundedly.
- Layer 4 (defer-recover) catches any panic that escapes the validators (defense against future lattigo changes).

The performance cost is negligible: the frame walker walks the wire bytes once, in $\mathcal{O}(n)$ where n is the byte count, and the lattigo deserializer would walk the same bytes anyway. The only added overhead is the explicit length-prefix bound checks.

6.5 Upstream reporting

The bug is in `luxfi/lattice` (Lux's pinned fork of `tuneinsight/lattigo` v7). On 2026-03-03 the Lux team filed the disclosure upstream as `luxfi/lattice` issue #2 ("DoS: ReadUint64Slice unbounded recursion + missing length cap") with the regression seed `testdata/fuzz/FuzzPulseDeserialize/c` from `github.com/luxfi/warp`. The fix landed in PR #3 ("security(buffer): bound ReadUintNSlice + iterative rewrite"), which:

- replaces the recursion in `ReadUint{16,32,64}Slice` with an O(1)-stack iterative loop;
- returns a new sentinel `ErrZeroProgress` when a Peek decodes zero elements while the destination still has capacity (the exact zero-progress recursion identified in §6);
- adds `Read{Uint16,Uint32,Uint64}SliceBounded` length-prefix-aware wrappers gated by `MaxSliceLen()` (default 1 MiB, configurable via the `LUX_LATTICE_MAX_UINT64_SLICE_LEN` environment variable).

Production deployments use the hardened `warp/pulsar/pulsar.go` validator stack regardless of upstream lattigo behavior, so a downstream lattigo change cannot regress the fix; the upstream patch is defense-in-depth for non-warp callers (ringqp deserialization, ciphertext unmarshalling, multiparty key shares).

6.6 Test surface

The fuzz harness continues to run in CI on every commit. The seed corpus includes:

- A valid Pulse from a real 3-of-2 ceremony.
- Empty bytes.
- Single `0x00` byte.
- Repeated `0xFF` bytes (1024 bytes).
- Truncated Pulse prefixes (1, 7, 15 bytes from a real Pulse).
- The original $\sim$ 30-byte exploit input.

`FuzzPulseDeserialize`, `FuzzPulseSerialize`, `FuzzHorizonCertificate`, and `FuzzWarpEnvelopeV2` run for 60 s in CI; all four return zero panics on the v0.1.0-rc1-pq-consensus-freeze tag. The 30-byte exploit is in the corpus as a regression test; it returns a clean error and does not panic.

6.7 Lessons

- **Cross-language deserializers are a structural risk.** The lattigo wire format originated in a research library; the destination chain runs untrusted bytes through it. Bound everything; trust nothing.
- **Fuzz harnesses pay for themselves.** The cost of writing the harness was hours; the cost of a destination-chain DoS in production would be days of incident response plus reputation damage.
- **Domain separation is necessary but not sufficient.** The fuzz finding is unrelated to the cryptographic soundness of the Warp 2.0 envelope shape (Section 5); it is a wire-format DoS. Both surfaces need attention.

The Mar-3 freeze ships with all four layers active. Subsequent fuzz finds are tracked in the same harness; new corpus entries are added on every find.

7 Implementation

The shipping implementation lives in two Go packages: `github.com/luxfi/warp` (the root package) and `github.com/luxfi/warp/pulsar` (the Pulse path). The split exists so the root package does not import the Pulsar kernel directly; the dispatch surface is the small `PulseVerifier` interface, with the concrete kernel-driven verifier in the subpackage.

7.1 Repository layout

Path	Role
`warp/envelope.go`	`EnvelopeV2`, `ParseEnvelope`, `ParseEnvelopeV2`, the version-byte dispatcher, and the wire-size guards
`warp/message.go`	v1 `Message` (unchanged)
`warp/signature.go`	v1 `BitSetSignature` (unchanged)
`warp/validator.go`	v1 validator-set / canonical validator-set machinery (unchanged)
`warp/verifier.go`	v1 `Verifier` interface (unchanged)
`warp/fuzz_envelope_test.go`	`FuzzWarpEnvelopeV2`: the version-byte dispatcher fuzz harness
`warp/pulsar/pulsar.go`	`KernelVerifier`, `BuildSigningBytes`, frame walkers, hardened deserializer
`warp/pulsar/classification.go`	`HorizonCertificate` helper (lifts envelope + transcript fields into the LP-105 cert shape)
`warp/pulsar/fuzz_pulse_test.go`	`FuzzPulseDeserialize`, `FuzzPulseSerialize`: the Pulse wire-format fuzz harness
`warp/pulsar/fuzz_horizon_cert_test.go`	`FuzzHorizonCertificate`: the Horizon cert fuzz harness
`warp/pulsar/groth16_classification_test.go`	Tests that the optional Groth16 wrapper is correctly classified as classical (`IsPQRootOfTrust` returns false)
`warp/pulsar/hashsuite_mismatch_test.go`	Tests `ErrSuiteMismatch` when the resolver-supplied suite differs from the envelope's `HashSuiteID`

7.2 The dispatch surface

The root `warp` package exposes `ParseEnvelope`: the dispatcher that detects v1 vs v2 wire bytes and returns an `EnvelopeV2`.

```
func ParseEnvelope(b []byte) (*EnvelopeV2, error) {
    if len(b) == 0 {
        return nil, ErrEnvelopeEmpty
    }
    switch b[0] {
    case EnvelopeVersion2:
        return ParseEnvelopeV2(b)
    default:
        // Fall back to v1; lift v1 message into a v2 envelope.
        msg, err := ParseMessage(b)
        if err != nil {
            return nil, fmt.Errorf("warp: %w", err)
        }
        return &EnvelopeV2{Message: msg}, nil
    }
}
```

The dispatcher is the only place where v1 / v2 wire formats are distinguished. Downstream consumers see a uniform `EnvelopeV2`; v1-only callers see a v2 envelope with PQ lanes empty.

7.3 The PulseVerifier interface

```
type PulseVerifier interface {
```

```
    VerifyPulse(env *EnvelopeV2) error
}
```

The root `warp` package's `VerifyV2` accepts a `PulseVerifier`. Production code wires `warp/pulsar/KernelVe` (the concrete Pulsar-kernel-driven verifier). Tests wire a `FakePulseVerifier` that records calls without invoking the kernel.

The split keeps the import graph clean: the root `warp` package does not import `github.com/luxfi/pulsar`, so it has no dependency on the lattigo deserializer; downstream consumers that want only the Beam path do not pay for the lattigo dependency.

7.4 The KernelVerifier flow

1. Parse the envelope (`ParseEnvelope`). v1 bytes lift to a Beam-only envelope.
2. If the envelope's `PulsarPulse` is empty, return early (Beam-only path).
3. Resolve (sourceChainID, sourceKeyEraID, sourceGeneration) via the `GroupKeyResolver` to obtain (GroupKey, HashSuiteID).
4. Check `HashSuiteID` matches the envelope's `HashSuiteID`; return `ErrSuiteMismatch` on mismatch.
5. Build the canonical signing bytes via `BuildSigningBytes`: H_T over (`WARP-PULSAR-ENVELOPE-v1`, `sourceChainID`, `payload`, `sourceNebulaRoot`, `sourceKeyEraID`, `sourceGeneration`).
6. Validate the Pulse wire bytes via the four-layer guards (Section 6): outer wire-size cap, per-lane frame cap, frame walker, defer-recover.
7. Deserialize via `DeserializePulse` into a `pulsarKernel.Signature`.
8. Verify via Pulsar.Verify(GroupKey, signingBytes, signature). Return the kernel's verification result.

The Beam verification (BLS aggregate) is run separately by the destination's existing v1 `Verifier`; the Pulse path is additive. A destination chain that wants the full Horizon path on its incoming envelopes wires both verifiers and Prism-binds them via the destination's own `HorizonCertificate` construction.

7.5 Test status

The Mar-3 freeze tag (`v0.1.0-rc1-pq-consensus-freeze` on `github.com/luxfi/warp`) carries the following test status:

- `warp/envelope_test.go`: 100% pass. v1 / v2 dispatcher coverage; round-trip; forward-compat decode of v1 bytes; backward-incompat reject of v2 bytes by v1 path.
- `warp/envelope_negative_test.go`: 100% pass. Malformed v2 envelope rejection.
- `warp/fuzz_envelope_test.go`: zero panics in 60 s.
- `warp/pulsar/pulsar_test.go`: 100% pass. End-to-end ceremony + sign + serialize + deserialize + verify.
- `warp/pulsar/fuzz_pulse_test.go`: zero panics in 60 s.
- `warp/pulsar/fuzz_horizon_cert_test.go`: zero panics in 60 s.
- `warp/pulsar/groth16_classification_test.go`: 100% pass. `IsPQRootOfTrust` correctly classifies Groth16-wrapped lanes as classical.
- `warp/pulsar/hashsuite_mismatch_test.go`: 100% pass. `ErrSuiteMismatch` returned on mismatch.

The build / test gate is `go test -count=1 ./...` from the `warp/` root.

7.6 Performance

Operation	Time
v1 envelope parse + verify (BLS aggregate, $n = 21$)	$\sim 875\,\mu$s
v2 envelope parse (no Pulse)	$\sim 30\,\mu$s
v2 envelope parse + Pulse deserialize + verify ($K = 21$)	~ 5 ms
Frame walker (`validatePolyFrame`) on 33 KB Pulse	$\sim 50\,\mu$s
`HashSuiteID` resolver lookup (cached)	$\sim 1\,\mu$s

The Pulse verify dominates v2 envelope verification cost; the Beam aggregate verify is a few-percent overhead. Measurements on Apple M1 Max single core; full numbers and reproduction commands in the benchmark report [4].

7.7 Cross-implementation parity

The Go canonical ([15]) is the byte oracle. The wire format is normative; any other-language implementation must produce byte-equal envelopes. The KAT vectors gate every release: fuzz seeds + a representative real-ceremony Pulse + the Mar-3 exploit input, all checked into `warp/pulsar/testdata/`.

7.8 Forward / backward compatibility tests

- `TestParseEnvelopeForwardCompatV1Bytes`: v2 receiver decodes v1 bytes, lifts into v2 envelope with empty PQ lanes, verifies via Beam-only path.
- `TestParseEnvelopeBackwardIncompatV2Bytes`: v1 receiver rejects v2 bytes (the leading `0x02` fails RLP-list prefix check).
- `TestEnvelopeRoundTrip`: v2 `Bytes()` → `ParseEnvelopeV2` produces a byte-equal envelope.
- `TestEnvelopeBeamUnchanged`: the BLS aggregate inside an `EnvelopeV2` is bit-equal to the BLS aggregate in a corresponding v1 `Message`.

All pass on the freeze tag.

8 Related Work

We compare Warp 2.0 to four cross-chain messaging protocols: Chainlink CCIP, Wormhole, IBC, and LayerZero. The structural lens is the cryptographic basis for source-chain finality: each protocol's source-finality story has a different shape, and Warp 2.0's contribution is making that story *post-quantum* via the Pulse, not classical-only via a BLS aggregate or oracle attestation.

8.1 Chainlink CCIP

CCIP [17] couples a Chainlink-operated oracle network with a separate Risk Management Network (RMN) for cross-chain message attestation. The cross-chain envelope is signed by the oracle network; the RMN provides a separately signed risk-management attestation.

Source-finality basis. Oracle attestation. The destination chain trusts that the oracle network correctly observed the source chain's finality and signed accordingly. The trust model is the union of the oracle network's classical signature scheme and the RMN's classical signature scheme.

Differences vs Warp 2.0. (1) CCIP's source finality is oracle-attested; Warp 2.0's is source-validator-signed (the source's own validator set produces the Pulse). (2) CCIP is classical-only; Warp 2.0 is hybrid PQ via the Pulse. (3) CCIP composes two oracle networks (Chainlink +

RMN); Warp 2.0 composes three signing lanes (Beam + ML-DSA + Pulse) on the same source-validator set.

Where they could meet. A CCIP-style oracle-attested envelope could carry a Pulsar Pulse attesting the oracle's view; the destination would verify the oracle's PQ-signed view rather than its classical-signed view. The integration is straightforward; the open work is on the CCIP side, not Warp 2.0's.

8.2 Wormhole

Wormhole [19] uses a 19-validator "guardian" network that signs cross-chain Verified Action Approvals (VAAs). The destination chain verifies the guardian quorum signature against the Wormhole guardian-key registry.

Source-finality basis. Guardian-network attestation. The destination trusts that the guardians correctly observed the source's finality and signed.

Differences vs Warp 2.0. (1) Wormhole's source finality is guardian-attested (a separate 19-of-13 committee); Warp 2.0's is source-validator-signed (the source chain's own validator set produces the Pulse, no separate committee). (2) Wormhole is classical-only (Ed25519 guardian keys); Warp 2.0 is hybrid PQ. (3) Wormhole's guardian set is fixed-size and externally curated; Warp 2.0's source-validator set is whatever the source chain runs, with LSS-managed dynamic resharing.

Wormhole 2.x and PQ. Wormhole has signaled intent to add PQ signatures to the guardian quorum. The shape would be analogous to Warp 1.x $\rightarrow$ 2.0 evolution. The structural difference remains: Wormhole's source finality routes through a separate guardian set; Warp 2.0's routes through the source's own validators.

8.3 IBC

IBC [2] (Inter-Blockchain Communication) uses light clients on the destination chain that follow the source chain's consensus headers. Source finality is verified by the destination running a verifier of the source's consensus protocol.

Source-finality basis. Direct consensus verification. The destination's IBC light client tracks the source's consensus state (Tendermint's prevote / precommit aggregate) and accepts cross-chain envelopes that the source has finalized.

Differences vs Warp 2.0. (1) IBC requires the destination to run a full source-consensus verifier; Warp 2.0 requires only a GroupKeyResolver keyed by source (η, g). (2) IBC is classical-only (Tendermint Ed25519); Warp 2.0 is hybrid PQ. (3) IBC's verification cost on the destination scales with the source's validator set size and per-block signature verification; Warp 2.0's verification cost is constant in the source's validator set size (one Pulse verify, one BLS aggregate verify, one ML-DSA cert-set verify or one Groth16 verify).

The light-client tradeoff. IBC's tightest binding to source finality is a strength: the destination can verify any source-finality property without trusting an oracle. The cost is operational: every source chain requires a separate IBC light client implementation on every destination. Warp 2.0 trades the tightest binding for a uniform GroupKeyResolver interface; the destination only needs to know the source's GroupKey lineage, not its full consensus rules.

8.4 LayerZero

LayerZero [20] couples an oracle (delivering source block headers) with a relayer (delivering cross-chain message proofs against those headers). The destination chain accepts a message iff the oracle and relayer agree.

Source-finality basis. Oracle + relayer agreement. The destination trusts that the oracle correctly observed the source's finality and that the relayer's proof is valid against the oracle's header.

Differences vs Warp 2.0. (1) LayerZero's source finality routes through two separate trusted parties; Warp 2.0's routes through the source's own validator set. (2) LayerZero is classical-only; Warp 2.0 is hybrid PQ. (3) LayerZero's trust model is two-of-two oracle / relayer; Warp 2.0's is the source's own $\geq 2/3$-weight validator quorum.

8.5 Comparison table

Protocol	Source-finality basis	PQ?	Trust model
CCIP	Oracle network + RMN	no	Oracle network classical sigs $\cup$ RMN classical sigs
Wormhole	19-validator guardian network	no	Guardian quorum classical sigs
IBC	Direct source-consensus verifier (light client)	no	Source's classical consensus correctness
LayerZero	Oracle (block header) + relayer (message proof)	no	Oracle classical sigs $\cap$ relayer correctness
Warp 1.x	**Source validator BLS aggregate**	**no**	**Source's $\geq 2/3$-weight BLS**
Warp 2.0	**Source validator Pulse + Beam + ML-DSA cert set**	**yes (hybrid)**	**Source's $\geq 2/3$-weight Pulse + Beam + ML-DSA, conjunctive Prism predicate**

The Warp 2.0 row is the only design that makes source-finality post-quantum and routes through the source's own validators. The combination is the contribution.

8.6 Where Warp 2.0 sits in the cross-chain design space

Warp 2.0's design space coordinates:

- **Trust routing**: source validators (Warp 1.x / 2.0) vs separate committee (Wormhole) vs oracle network (CCIP, LayerZero) vs direct light-client verification (IBC).
- **Cryptographic basis**: classical aggregate (BLS, Ed25519) vs hybrid PQ (Pulse + Beam + ML-DSA).
- **Source-finality binding**: NebulaRoot + KeyEra/Generation lineage (Warp 2.0) vs block header (IBC) vs payload-only (Warp 1.x, CCIP, Wormhole, LayerZero).

Warp 2.0's coordinates are: source validators, hybrid PQ, NebulaRoot + lineage. The space is not exhaustively covered; the four neighbors illustrate the trade-offs but do not bound the design space. A future protocol could combine, e.g., a source-validator routing with a STARK proof of finality (no GroupKeyResolver needed); the open work is the production deployment of such systems, and the conditions under which they outperform Warp 2.0.

8.7 Status

Warp 2.0 ships in the Mar-3 freeze. Warp 1.x continues to ship for backwards compatibility. The wire is forward-only; the two channels coexist without negotiation. Cross-chain destinations on Lux mainnet (Lux → Hanzo, Lux → Zoo, etc.) verify v2 envelopes against the source chain's Pulsar GroupKey lineage as recorded in their on-chain source-key-registry contracts. The PaaS deployment path [16] updates the registry contracts as the source chains' Reanchor lifecycles dictate; no out-of-band oracle is required.

References

[1] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In *ASIACRYPT*. Springer, 2001.

[2] Cosmos Network. Inter-blockchain communication protocol (IBC). https://ibcprotocol.org, 2021.

[3] Lux Industries Inc. luxfi/p3q: Post-quantum plonky3 with sha3 challenger. https://github.com/luxfi/p3q, 2026.

[4] Zach Kelling. Lux post-quantum consensus benchmarks. Lux Industries, Inc.; `papers/lux-pq-consensus-benchmarks/`, 2026.

[5] Zach Kelling. Post-quantum finality for lux quasar. Lux Industries; `lux/proofs/pq-finality-no-bls.tex`, 2026.

[6] Zach Kelling. Pulsar: Dynamic lattice threshold signatures for permissionless validator sets. Lux Industries, Inc.; `papers/lp-073-pulsar/`, 2026.

[7] Zach Kelling. Quasar horizon: Hybrid post-quantum finality over dag-native nebula roots. Lux Industries, Inc.; `papers/lux-quasar-horizon/`, 2026.

[8] Zach Kelling. QuasarCert soundness: Canonical formal specification, quasar 3.0. Lux Industries; `lux/proofs/quasar-cert-soundness.tex`, 2026.

[9] Lux Core Team. Lattigo v7: A lattice-based cryptography library (lux fork). https://github.com/luxfi/lattice, 2026. Fork of `tuneinsight/lattigo`; pinned at v7 for byte-exact reproducibility.

[10] Lux Core Team. Lp-020: Quasar consensus 3.0. https://github.com/luxfi/lps/blob/main/LP-020-quasar-consensus.md, 2026.

[11] Lux Core Team. Lp-021: Warp cross-chain messaging (1.x). https://github.com/luxfi/lps/blob/main/LP-021-warp-messaging.md, 2026.

[12] Lux Core Team. Lp-073: Pulsar — dynamic lattice threshold signatures for permissionless validator sets. https://github.com/luxfi/lps/blob/main/LP-073-corona.md, 2026.

[13] Lux Core Team. Lp-075: Bls aggregate signatures for warp messaging. https://github.com/luxfi/lps/blob/main/LP-075-bls-aggregate.md, 2026.

[14] Lux Core Team. Lp-077: Linear shamir's secret sharing — universal threshold lifecycle. https://github.com/luxfi/lps/blob/main/LP-077-linear-shamir.md, 2026.

[15] Lux Core Team. Warp: Cross-chain messaging protocol (go canonical). https://github.com/luxfi/warp, 2026.

[16] Lux Industries. LP-105: The lux finality stack — public vocabulary and internal names. Lux Improvement Proposal, 2026.

[17] Sergey Nazarov et al. Chainlink cross-chain interoperability protocol (CCIP). In *Chainlink Whitepaper*, 2023.

[18] Vishnu J. Seesahai and Zach Kelling. LSS: A universal lifecycle framework for threshold cryptographic protocols. Lux Industries, Inc., 2026.

[19] Wormhole Foundation. Wormhole: Cross-chain messaging protocol. https://wormhole.com, 2022.

[20] Ryan Zarick, Bryan Pellegrino, and Caleb Banister. Layerzero: An omnichain interoperability protocol. https://layerzero.network/whitepaper, 2022.