# Lux MPTC Side-Channel Considerations

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.** The side-channel considerations summary for the NIST MPTC submission. Detailed reviews live in the canonical `CONSTANT-TIME-REVIEW.md` files in each kernel repo (`pulsar/CONSTANT-TIME-REVIEW.md`, `lens/CONSTANT-TIME-REVIEW.md`); this document summarizes the posture and points at the production-stack hardening.

## 1 Constant-time review (Pulsar)

**Reference.** `pulsar/CONSTANT-TIME-REVIEW.md`.

**Surface covered.**

- Lattice ring arithmetic ($R_q$ multiplication, NTT / iNTT, Montgomery reduction).
- Discrete-Gaussian samplers (Ziggurat tables, BLAKE3-XOF buffer-refill).
- Pulsar Sign1, Sign2, Combine, Verify — the core threshold signing protocol.
- Lattice Pedersen commit / verify (used in resharing).
- Pairwise PRF / MAC (KMAC256-based, constant-time per FIPS 202 hardware).

**Posture.** The lattice math is constant-time at the Go-source level: no data-dependent branches, no data-dependent memory access patterns, no early termination based on secret values. The Lattigo `v7` fork ([3]) inherits this posture from the upstream Lattigo project, with Lux-specific additions reviewed.

**What is not covered.**

- Go runtime garbage-collection-induced timing variance: fundamental to Go; mitigated by tuning GOGC for deployments that need stricter constant-time guarantees.
- Cache-timing on heterogeneous hardware: varies across deployment targets; Lux production stack runs on bare-metal x86 / ARM and assumes a benign cache-timing posture.
- Constant-time GPU code paths: GPU code is research, not production.

## 2 Constant-time review (Lens)

**Reference.** `lens/CONSTANT-TIME-REVIEW.md`.

**Surface covered.**

- Curve scalar multiplication on Ed25519, secp256k1, Ristretto255 (constant-time per curve).
- FROST 2-round Schnorr threshold signing (Sign1, Sign2, Aggregate, Verify).
- Curve Pedersen commit / verify.

**Posture.** The curve math uses upstream constant-time implementations (`golang.org/x/crypto/ed25519`, `decred/dcrd/secp256k1`, `gtank/ristretto255`) without modifications that would compromise constant-time. The Lux-specific additions (key-era lifecycle wrapping, LSS adapter) operate on already-derived public values and do not introduce new secret-dependent code paths.

## 3 Lattigo deserializer hardening

**Surface.** The lattigo `ReadUint64Slice` function recurses on length prefixes without bounding by available bytes. An attacker-crafted Pulse can trigger unbounded recursion and DoS the destination chain.

**Discovery.** Found by the Mar-3 fuzz harness at `warp/pulsar/fuzz_pulse_test.go` (FuzzPulseDeserialize) within a 60-second fuzz run. The exploit is a $\sim$ 30-byte input.

**Mitigation.** Four-layer defense in `warp/pulsar/pulsar.go`:

1. `MaxPulseWireSize = 64 KB`: outer wire-size cap.
2. `MaxPulseFrameSize = 32 KB`: per-lane (C, Z, $\Delta$) cap.
3. `validatePolyFrame` / `validateVectorPolyFrame`: end-to-end frame walker that checks every length-prefix against `MaxLatticeUintSliceLen = 4096` before lattigo's `ReadFrom` is invoked.
4. `defer recover()`: defense-in-depth boundary that converts any escaping panic into a clean error.

**Test coverage.** The original $\sim$ 30-byte exploit is in the fuzz corpus as a regression test; `FuzzPulseDeserialize`, `FuzzPulseSerialize`, `FuzzHorizonCertificate`, and `FuzzWarpEnvelopeV2` return zero panics in 60-second CI runs on the freeze tag. Full discussion: [2] Section "Fuzz-Discovered Vulnerability".

## 4 Timing side-channels in production

**Network timing.** Pulse signing is a 2-WAN-crossing protocol (Sign1 + Sign2). The chain's state machine does not act on signing-protocol timing differences; the consensus engine accepts a Pulse iff Pulsar.Verify succeeds, regardless of how long signing took. A network adversary who learns Pulse signing latency does not learn share secrets.

**Verifier timing.** The Pulsar verifier is constant-time at the Go-source level; its runtime depends on the public verification predicate ($\|z\|_2^2 \le B^2$, $c$ recomputation, $\mathbf{A} \cdot z = D + c \cdot \tilde{\mathbf{b}} + \Delta$), not on secret values. A timing oracle on the verifier reveals public-input structure (signature size, validator-set size) but not share material.

**Memory access patterns.** The lattice ring arithmetic uses fixed-size buffers; vector and matrix operations iterate over their full domains regardless of the secret values they contain. Cache-timing leakage is bounded by the structural memory footprint, which is public.

## 5 Power side-channels

The Lux production stack runs on commodity cloud hardware (DigitalOcean DOKS K8s pods); power-side-channel attacks require physical access. The threat model excludes physical-access attacks.

For deployments that run on private hardware (bare-metal validators), power-side-channel mitigations are deployment configuration: TEMPEST shielding, physical access controls, and HSM-backed key custody. The Lux stack supports HSM-backed key custody via the standard signer interface (`pulsar/sign/Signer`, `lens/sign/Signer`); an HSM that holds shares behind a hardware boundary mitigates the power-side-channel surface to whatever the HSM provides.

## 6 Fault-injection side-channels

**Surface.** An adversary inducing faults during signing (rowhammer, voltage glitching) could force the Pulsar Sign2 path to produce malformed partial signatures that aggregate into a forgery.

**Mitigation.** The activation cert path (Section "Pulsar.Activate" of [1]) tests the new shares against the unchanged GroupKey before accepting a generation transition; a fault-injected share that does not actually correspond to the GroupKey will fail activation and trigger rollback.

For per-Pulse fault-injection, the four-layer Pulse hardening (Section "Lattigo deserializer hardening") catches malformed wire bytes; the Pulsar verifier itself is structurally fault-tolerant via the $L_2$ acceptance bound and the rounding-hint correction. A successful per-Pulse fault-injection attack would need to forge an input that passes both the wire-format validators and the lattice acceptance predicate; we have no evidence of such an attack and do not claim formal fault-injection resistance.

## 7 Failure-mode containment

LSS's no-slashing-dependency rollback ladder ([4]) is a structural mitigation against side-channel-induced share compromise. If a side-channel attack reveals enough share information to forge an activation cert, the chain's safety holds because the activation cert verification step itself catches the forgery (Theorem "Activation cert is necessary and sufficient" of `proofs/pulsar/activation-safety.tex`). If the attacker forges a routine bundle Pulse, the bundle stays at Pulse-sealed but the chain's other lanes (Beam, ML-DSA cert set) catch the inconsistency at the Prism layer.

## 8 What is out of scope

- Constant-time guarantees on GPU code paths (GPU production not enabled).
- Constant-time guarantees against the Go runtime's GC-induced timing.
- Power-side-channel resistance on commodity cloud hardware.
- Formal fault-injection resistance.

These are tracked as known limitations (`known-limitations.md`). Production deployments that require stricter side-channel posture should deploy on bare metal with HSM-backed key custody and disable the GC-induced variance via Go runtime tuning.

## 9 Review status

| Component | Review status |
|---|---|
| Pulsar lattice math (Sign / Verify / Combine) | Internal constant-time review complete |
| Pulsar lifecycle (Bootstrap / Reshare / Refresh / Activate) | Internal constant-time review complete |
| Pulsar DKG2 (Pedersen DKG over $R_q$) | Internal review complete; production-research |
| Lens curve math (Sign / Verify / Combine) | Internal constant-time review complete |
| Lens lifecycle | Internal constant-time review complete |
| LSS framework | Internal review complete |
| Quasar Horizon composition | Internal review complete |
| Warp 2.0 envelope | Internal review complete + Mar-3 fuzz hardening |
| GPU paths | Constant-time review NOT complete; production NOT enabled |
| External third-party audit | Open work; production-research roadmap |

## 10 Conclusion

The CPU production path on the freeze tag has been through internal constant-time review for both the lattice (Pulsar) and curve (Lens) kernels. The lattigo deserializer is hardened against attacker-controlled length-prefix DoS via four-layer defense. The fuzz harnesses run continuously and have surfaced one production vulnerability (the lattigo `ReadUint64Slice` DoS) which is fixed.

External third-party audit and GPU constant-time review are open work. Production deployments that need stricter side-channel posture should follow the deployment-configuration recommendations above.

## References


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

[2] Zach Kelling. Warp 2.0: Pulse-backed cross-chain source finality. Lux Industries, Inc.; `papers/lux-warp-v2/`, 2026.

[3] Lux Core Team. Lattigo v7 (lux fork). https://github.com/luxfi/lattice, 2026.

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