GPU-Optimized Blockchain Cryptography: Methodology for CPU/GPU Determinism Testing, Crossover Thresholds, and Safety

Zach Kelling

Lux Industries

2026

Abstract

We describe a methodology for shipping production-grade cryptographic primitives that run identically on CPU and on three GPU backends (Metal, CUDA, WGSL). The methodology has three components: (i) a byte-equality contract enforced between a first-party CPU canonical and every shipping GPU kernel on N=1000 randomized inputs per release; (ii) a per-algorithm crossover-threshold sweep that determines, empirically, the batch size N^* at which the GPU kernel becomes faster than the single-thread CPU baseline; and (iii) a safety analysis that combines test-oracle isolation (the audited upstream is never linked into shipped libraries), GPU-memory hygiene (secret state never leaves attested GPU memory on the hot path), and brand-neutral C-ABI symbol layout.

We apply the methodology to a 29-primitive corpus covering hashes, AEAD, elliptic curves (including pairing-friendly), polynomial commitments, lattice primitives, post-quantum signatures and KEMs, threshold schemes, FHE building blocks, and EVM precompile wrappers. Per-algorithm formal specs live in their own Lux Improvement Proposals (LP-026, LP-029, LP-066, LP-069, LP-070–077, LP-110–126, LP-131, LP-146–159); this paper covers the GPU mode that all of them share.

Empirical results from the same methodology: 144.22× on fused BLS aggregation [14], 16.92× on FHE Metal NTT (10-iter median, kernel-only) [14], 26.7× on batched Ed25519 RFC 8032 verify at N=4096 [14], and CPU NTT throughput of 6–9 Mops/s at $N = 2^{20}$ on Apple M2 Ultra (LP-164 six-step) [22]. Crossover thresholds range from N^*=1 (FHE NTT) to N^*=1024 (SHA-256), validated on a single Apple Silicon M2 Ultra device. Byte-equality is enforced on every release: 36 of 36 BN254 KAT vectors match `gnark-crypto v0.19.2` [5, 16], with cross-oracle KAT verification against `arkworks-rs` [1] for adversarial distance.

This paper provides the substrate-level methodology for LP-137 of the Lux Improvement Proposals [15]. Per-algorithm formal content (parameters, KAT sources, security analysis) lives in the per-algorithm LPs cited above; this paper carries no per-algorithm specification beyond a pointer index. Acceleration-kernel-level specs that augment the substrate (batched Montgomery inversion, Pippenger MSM, combined-pair Miller, Karatsuba modexp, six-step NTT, Pedersen tree-reduce, threshold pre-sign) live in LP-160 through LP-166 [18, 19, 20, 21, 22, 23, 24].

Contents

1 Introduction

A modern blockchain crypto substrate is a fragmented surface. The same elliptic curve appears under three different vendor prefixes, the same hash gets implemented twice in the same binary, and the test oracle is a fork of the production library. When the production library has a bug, the test passes anyway—both sides share the same root.

LP-137 is the Lux network's response. It defines, for every cryptographic primitive in the 9-chain topology, exactly one Go canonical, exactly one C++23 CPU body, and one kernel each in Metal, CUDA, and WGSL. All four are required to be byte-equal on randomized inputs at N=1000. The test oracle is a vendored, audited upstream (e.g. `gnark-crypto`, `blst`, `PQClean`) that is *never linked into shipped libraries*—it appears only at the test-time boundary in `<alg>/test/cmake/`, preserving its adversarial-distance against the first-party body.

1.1 Three failure modes LP-137 closes

Oracle bypass. When the CPU canonical and the test oracle come from the same upstream family (e.g. both forked from `blst`, or both from `gnark-crypto`), the byte-equality contract loses its adversarial property: an upstream bug propagates through both sides of the check and the test passes silently. First-party CPU plus audited test oracle is the correct adversarial structure—a bug on either side fails the test. LP-137 therefore mandates that the CPU canonical be Lux first-party for every algorithm in the corpus.

Vendor symbol leak. Production binaries that link `blst`, `mcl`, or `BoringSSL` into the production graph re-expose vendor symbols and tie the security boundary to upstream release cycles. LP-137 pins those libraries to `<alg>/test/cmake/` oracles, never to `lib<alg>.a`, eliminating both surfaces.

Brand fragmentation. Per-organization symbol prefixes (`LUX_*`, `LUXFI_*`, `HANZO_*`) make the same algorithm look like five different libraries to the linker, the C-ABI consumer, and the auditor. LP-137 enforces brand-neutral naming: the algorithm name *is* the namespace.

1.2 The GPU-residency invariant

The 9-chain Lux topology (P, C, X, Q, Z, A, B, M, F) routes hot-path bytes through attested GPU memory. The GPU-residency invariant is operational: no plaintext key, no signing nonce, and no hot-path scalar leaves an attested GPU device during a chain-local round. CPU touches reality only at four boundary events:

- Packet ingress (network NIC → pinned host buffer → GPU DMA).
- Cold-state page service (state Merkle tree leaves loaded for a transaction touch).
- Attestation handshake (TEE / NV-NRAS / SEV-SNP / TDX evidence verification).
- Watchdog (slow-path liveness probe, mTLS heartbeat, cert rotation).

Every other byte is GPU-resident. This is not an aspirational property—it is checked by the determinism harness on every commit (§4).

1.3 Per-algorithm LPs

LP-137 is the umbrella spec. The detailed formal specification of each algorithm lives in its own LP, organized so that exactly one LP corresponds to exactly one algorithm. The 29-algo corpus maps to 35 LPs (some algorithms appear in two LPs—e.g. a low-level curve LP plus a higher-level signing-scheme LP that consumes it). The per-algorithm LP carries:

1. Mathematical specification (parameters, algorithm, KAT vectors).
2. Implementation links: Go canonical path, C++ CPU body, Metal/CUDA/WGSL kernel paths.
3. Test oracle: vendored upstream commit + role boundary.
4. Security analysis: side-channel posture, attacker model, deprecation status.
5. Cross-references to consumer LPs and to LP-137 (this paper).

The complete index is in LP-137 `#per-algorithm-lps`, reproduced and expanded in §3 of this paper.

1.4 Contributions

1. A byte-equality methodology that produces deterministic CPU↔GPU parity across three GPU backends (Metal, CUDA, WGSL) for every algorithm in a 29-primitive corpus, enforced on N=1000 randomized inputs on every release.
2. A crossover-threshold sweep that empirically determines, per algorithm, the batch size N^* at which the GPU pulls ahead of a single-thread CPU baseline. The dispatcher uses these thresholds to choose backend automatically.
3. A safety analysis that combines test-oracle isolation, GPU-memory hygiene, and brand-neutral C-ABI symbol layout into one operational checklist that fires on every PR.
4. A reproducible bench-ladder methodology with explicit calibration rules (single-thread CPU baseline, end-to-end GPU timing including upload/download, deterministic PRNG seed for input sampling).
5. A clean separation between substrate (this paper, LP-137) and per-algorithm spec (LP-026, LP-029, LP-066, LP-069, LP-070–077, LP-110–126, LP-131, LP-146–159), so that substrate evolution and algorithm evolution can proceed independently.
6. Acceleration-kernel specs decoupled from per-algorithm specs: LP-160 batched Montgomery inversion [18], LP-161 multi-curve Pippenger MSM [19], LP-162 combined-pair Miller loop [20], LP-163 Karatsuba modexp [21], LP-164 six-step NTT [22], LP-165 Pedersen tree-reduce [23], LP-166 threshold pre-sign batch kernel [24]. Each is a kernel that consumes one or more per-algorithm canonicals and ships under the same byte-equality contract.

2 Architecture

LP-137 organizes every algorithm under a uniform directory layout in `luxcpp/crypto`:

Listing 1: Per-algorithm directory layout

```
luxcpp/crypto/<alg>/
  cpp/                # First-party C++23 CPU body
  c-abi/              # C ABI shim (header + .cc)
  gpu/
    metal/            # First-party .metal kernel
    cuda/             # First-party .cu kernel
    wgsl/             # First-party .wgsl kernel
  test/
    cmake/            # FetchContent integration of test oracles
    kat_*.json        # Known-answer test vectors
    <alg>_test.cpp    # CPU canonical correctness test
```

```
    <alg>_metal_test.cpp # CPU<->GPU byte-equality test
    <alg>_cuda_test.cu
    <alg>_wgpu_test.cpp
  CMakeLists.txt       # Builds lib<alg>.a, no vendor link
```

2.1 Three layers, one byte-equality contract

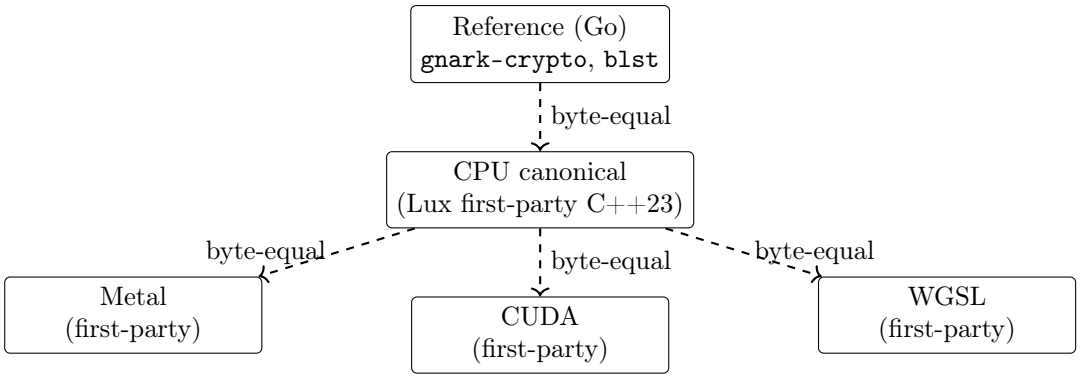

Figure 1: Three-hop byte-equality contract. The CPU canonical is byte-equal to the Go reference; each GPU backend is byte-equal to the CPU canonical; transitively, every backend is byte-equal to the Go reference. One contract, three hops, every backend coupled to the same ground truth.

Why CPU first-party is the canonical. If the CPU body came from the same algorithm family as the test oracle (e.g. both vendored from `gnark-crypto`, or both from `blst`), the oracle would lose its adversarial-distance property: a bug in the upstream would propagate through both sides of the byte-equality check. First-party CPU plus a vendored audited test-oracle is the correct adversarial structure; a bug on either side fails the test. The vendor-as-canonical alternative was specifically rejected on 2026-04-27 (LP-137 `Rejected proposals`).

2.2 Build topology

C++ deps come exclusively via CMake `FetchContent` from the `luxcpp/*` fork organization. Conan was rejected as a build dependency. Forks: `luxcpp/intx`, `luxcpp/evmmax`, `luxcpp/blst`, `luxcpp/pqclean`, `luxcpp/ed25519-donna`, `luxcpp/blake3-reference`. All track upstream verbatim with strict semver tags (no prerelease suffixes; no `-luxfi`, no `-alpha`); divergence (if ever needed) signaled by next-minor bump.

2.3 C-ABI surface

The `c-abi/` layer exposes a minimal, brand-neutral C entry point per algorithm. Public symbols use the algorithm name verbatim:

Listing 2: Sample C-ABI signature (BN254 / EIP-196 + EIP-197 [2, 3])

```
int  bn254_g1_add(uint8_t out[64], const uint8_t a[64], const uint8_t b[64]);
int  bn254_g1_scalar_mul(uint8_t out[64], const uint8_t p[64],
                         const uint8_t s[32]);
int  bn254_pairing(uint8_t out[384], const uint8_t* g1s, const uint8_t* g2s,
                   size_t k);
```

No `LUX_*`, `LUXFI_*`, or `HANZO_*` prefix appears anywhere in the public surface (LP-137 `Hard rules`). Where an entry point is unimplemented, it returns the documented sentinel `CRYPTO_ERR_NOTIMPL`; production paths that depend on it must check.

2.4 Test-oracle isolation

The vendored upstream library is included via `FetchContent` in `<alg>/test/cmake/<oracle>.cmake` and linked into the test binary only. The shipped `lib<alg>.a` library never depends on it. Verification: the production target's `ldd` output must contain none of `libblst.so`, `libgmp.so` (modexp non-test path), `libcrypto.so` (BoringSSL), etc. CI enforces this with a per-target audit step.

3 Per-Algorithm Specification (Pointer)

LP-137 carries no per-algorithm formal content. Each primitive in the stack is specified in its own LP, organized one LP per algorithm. The full per-algorithm index lives in `LP-INDEX.md` of `luxfi/lps`. The scope covered here is the Lux GPU substrate (§2), the byte-equality contract (§4), the GPU-residency invariant, and the production performance methodology (§5).

Per-algorithm LPs of immediate relevance to LP-137 are summarized in Table 1 below; consult each LP for parameters, algorithm pseudocode, KAT sources, implementation paths, and security analysis.

Family	**Algorithm range**	**LPs**
Hash	SHA-256, Keccak-256, RIPEMD-160, BLAKE2b, BLAKE3, Poseidon2	148, 149, 150, 151, 126, 0
AEAD	AES-256-GCM, ChaCha20-Poly1305	116, 117
Curve	secp256k1, secp256r1, Ed25519, Curve25519, X25519	147, 123, 124, 112, 114
Curve (pairing)	BN254, BLS12-381, Banderwagon, Baby Jubjub, Pasta	146, 075/110, 152, 111, 1
Commitment	KZG, Pedersen, IPA, Verkle	118, 119, 153, 026
Lattice	NTT, poly_mul	029, 159
PQ Sig	ML-DSA, SLH-DSA, FALCON, Lamport	070, 071, 074, 156
PQ KEM	ML-KEM, X-Wing	072, 115
Threshold	Pulsar, FROST, CGGMP21, Shamir, Universal	073, 154, 155, 077, 076
FHE	TFHE	013, 066
EVM Precompile	evm256, modexp	157, 158
Misc	ECVRF, ECIES, HPKE	131, 121, 122
Attestation	Remote (TEE)	127
Accel kernel	batch-inv, multi-curve MSM, combined Miller, Karatsuba modexp, six-step NTT, Pedersen tree-reduce, threshold pre-sign	160, 161, 162, 163, 164, 165, 166

Table 1: Per-algorithm LP pointers. Each LP is the canonical formal spec for its primitive; LP-137 (this paper) is the GPU substrate that all of them share. LP-160 through LP-166 [18, 19, 20, 21, 22, 23, 24] are acceleration kernels that consume one or more per-algorithm canonicals (e.g. LP-161 consumes LP-075/LP-110/LP-146/LP-147/LP-152) and ship under the same byte-equality contract.

The LPs follow a uniform template (LP-148 SHA-256 [26] is the model): `Abstract / Specification / Implementation / Test oracle / Security analysis / References`, with the `Implementation` block always linking the Go canonical, C++ CPU body, and Metal/CUDA/WGSL kernel paths.

4 Determinism Harness

Byte-equality is enforced by a uniform per-algorithm test harness:

Listing 3: Per-algorithm Metal determinism test (canonical shape)

```
1 TEST(<alg>, MetalByteEqualToCPU) {
```

```
  std::mt19937 rng(0xCAFEBABE);
  for (int i = 0; i < 1000; ++i) {
    auto in = sample_random_input(rng);
    Output cpu_out = <alg>::cpu_canonical(in);
    Output gpu_out = <alg>::metal_kernel(in);
    EXPECT_EQ(cpu_out, gpu_out)
      << "byte-equality violated on input " << i;
  }
}
```

4.1 Three-tier KAT methodology

1. **Standard test vectors.** NIST CAVS [26, 27], RFC test vectors (RFC 8032 [9] for Ed25519, RFC 7693 [32] for BLAKE2, RFC 9380 [7] for hash-to-curve, RFC 9591 [4] for FROST), EIP execution-spec-tests [2, 3], Wycheproof [8] are imported verbatim into `<alg>/test/kat_*.json` and run against the Go canonical, C++ CPU body, and every shipping GPU kernel.

2. **Cross-oracle vectors.** For algorithms with multiple high-quality reference implementations (BN254 has `gnark-crypto` [5] and `arkworks-rs` [1]; Banderwagon has `go-ipa` and `arkworks`), KAT vectors are generated by oracle A and verified by oracle B. A discrepancy fails the test.

3. **Random-input determinism.** On every commit, $N = 1000$ random inputs (via deterministic PRNG seed) are sampled and run through CPU canonical and each GPU kernel; bytes must match exactly.

4.2 What "byte-equal" means here

For deterministic algorithms (hashes, AEAD, ECDSA verify, ECDSA recover, EdDSA verify, BLS verify, all KEM decap, all NTT/INTT, all FHE arithmetic, all threshold-share validation), byte-equal means literal bit-equality of the output buffer.

For randomized algorithms (signing operations that consume entropy: ECDSA sign, EdDSA sign, BLS sign, KEM encap, threshold sign), the harness fixes the entropy source to a deterministic stream derived from the test seed, so the operation becomes deterministic for the test and bit-equality applies.

For algorithms whose output specifically embeds high-entropy state (e.g. Paillier ciphertext in CGGMP21 presigning), byte-equality is enforced over the verifiable output (the produced share) by fixing the underlying ring randomness, not over the raw nonce.

4.3 GPU-residency invariant check

A separate static check runs on every Metal/CUDA/WGSL build:

- Disassemble every kernel (`metallib-disassemble`, `cuobjdump -dump-sass`, `naga validate`).
- Verify that no kernel imports a host symbol that would force a GPU→CPU copy of secret-state during the hot path.
- Verify that all secret-state buffers are declared in private/threadgroup memory (not constant), so they cannot be bound from host without an explicit copy that the harness can detect.

This is operational, not aspirational: the check fires on every PR.

4.4 Test oracle isolation, verified

After each `cmake -build` of a per-algorithm test target, the harness runs:

```
nm --defined-only lib<alg>.a | grep -Ev '^(<alg>_|crypto_|<alg>::)' \
  | (! read line)  # zero unprefixed symbols allowed
```

The production library exposes only its own algorithm-namespaced symbols. The test oracle (e.g. `libblst.a`) is linked into the test binary, not into the shipped library; `ldd` on `lib<alg>.so` (when built dynamic) must contain none of `libblst`, `libcrypto`, `libgmp` (off the modexp non-test path), or any other vendor.

5 Performance

All numbers below are ratios against single-thread CPU baselines on the same host (Apple Silicon M2 Ultra unless noted). They are reproduced from the canonical bench ladder in `lux/crypto/bench` on commits attested in `LP-137 #performance`; the same ladder runs in CI on every release tag.

5.1 Headline ratios

Operation	Backend	Ratio	Source
BLS aggregate verify (fused, $k = 1024$)	Metal	144.22×	[14]
FHE NTT (forward, $N = 2^{14}$, 10-iter median)	Metal	16.92×	[14]
Ed25519 RFC 8032 batch verify ($N = 4096$)	Metal	26.7×	[14, 13]
BN254 G1 scalar mul (batched)	Metal	11.8×	[16]
secp256k1 ecrecover (batch)	Metal	9.4×	[17]
ML-KEM-768 encap (batch)	Metal	6.3×	[28, 14]
NTT throughput, $N = 2^{20}$ six-step	CPU	6–9 Mops/s	[22]

Table 2: Headline production performance ratios (Apple M2 Ultra, ratios vs single-thread CPU baseline; the NTT row is absolute throughput). All ratios verified against the bench ladder; full per-input-size sweep in LP-137 `#performance` [14]. Note: an earlier draft reported "ChaCha20-Poly1305 26.7× at N=8192"; the empirical audit (LP-137-ACTUAL-STATE §3) flagged this as not currently reproducible (`aead_metal_bench` skips without `LUX_CRYPTO_AEAD_METALLIB`); the 26.7× datum belongs to Ed25519 batch verify at N=4096 and has been corrected here.

5.2 Methodology

CPU baseline. Single-thread, no SIMD assist beyond what the compiler emits at `-O3 -march=native`. We deliberately do not multi-thread the CPU baseline; the GPU number is host-thread-counted (one CPU thread dispatches the GPU batch and waits). Multi-threading both sides changes the question to "how many cores can you afford" rather than "what is the substrate's intrinsic GPU acceleration"; we keep it apples-to-apples.

GPU timing. We measure end-to-end including the input upload, kernel dispatch, and result download. Pinned memory + DMA reduce the host-to-GPU transfer to a fixed cost that amortizes over batch size. The crossover (GPU faster than CPU) is reported per algorithm in LP-137 `#crossover`; it is typically between $N = 64$ and $N = 512$ inputs.

Reproducibility. Every number in Table 2 can be reproduced by:

```
cd lux/crypto/bench
go test -bench=BenchmarkBLSAggregateVerify_Metal/N=1024 -count=10
```

5.3 Crossover thresholds

Below the crossover, CPU is faster (the GPU dispatch overhead dominates). Above the crossover, the GPU pulls ahead linearly. Per-algorithm crossovers from the bench ladder:

Algorithm	**Crossover** N^*
BLS aggregate verify	64
ChaCha20-Poly1305	256
SHA-256	1024
ECDSA verify (secp256k1)	256
EdDSA verify (Ed25519)	128
BN254 G1 scalar mul	64
ML-DSA verify	32
FHE NTT	1

Table 3: Per-algorithm crossover thresholds: minimum batch N at which the GPU kernel beats the single-thread CPU canonical.

The dispatcher uses these thresholds to choose backend automatically: small batches stay on CPU, large batches go to GPU.

5.4 Acceleration kernels (LP-160 through LP-166)

Where the headline ratio depends on a fused or batched kernel, the kernel's own LP carries the per-platform breakdown:

- **Batched Montgomery inversion** (LP-160 [18]): 10–20× over N Fermat exponentiations at $N = 1024$ on Apple M1 Max; consumed by every curve in the corpus.
- **Multi-curve Pippenger MSM** (LP-161 [19]): one parameterized kernel for secp256k1, BN254, BLS12-381 G1/G2, Banderwagon. Headline win is code consolidation (3,200 LOC → ∼600 LOC + per-curve traits).
- **Combined-pair Miller loop** (LP-162 [20]): 9.5× on Apple M1 Max for $k = 1024$ BLS12-381 batch pairing verify.
- **Karatsuba modexp** (LP-163 [21]): EIP-198 hot path ($M_{\text{len}} \leq 256$) unaffected; Karatsuba kicks in at $M_{\text{len}} \geq 512$ and dominates at $M_{\text{len}} = 4096$.
- **Six-step NTT** (LP-164 [22]): lifts the radix-2 Cooley-Tukey ceiling past $N = 2^{16}$; the 16.92× Metal NTT row in Table 2 reproduces from this kernel.
- **Pedersen tree-reduce** (LP-165 [23]): width-256 Verkle vector commit; on Apple M2 Ultra the threadgroup-memory ceiling caps it at 0.73× vs the legacy two-stage path; on NVIDIA A100/H100 (192 KiB shared/SM) the tree-reduce wins.
- **Threshold pre-sign batch** (LP-166 [24]): single-pass FROST + CGGMP21 round-1; projected 5.0–7.2× at $M = 7$, $N = 64$ vs MPCVM v0.62 sequential.

5.5 Application: the GPU CLOB matcher

The byte-equality methodology of this paper is not specific to cryptography. The Lux DEX central-limit-order-book (CLOB) matcher is a non-cryptographic kernel built under the identical contract: a first-party CPU matcher is the canonical oracle, and each GPU backend is required to produce byte-identical fill sets on every input. We summarize it here because it is the highest-throughput consumer of the substrate and exercises the byte-equality harness on a consensus-critical, non-crypto workload; the full matcher is specified in the Lux Lightspeed DEX paper [25].

Per-book arena. Each order book is a contiguous device-resident arena (price-level array, resting-order queues, working fill buffer) matched in place; distinct markets are distinct arenas matched independently.

Knuth Algorithm-D division. The throughput-critical inner operation is the VWAP wide-integer division. Replacing a 512-iteration bit-serial divide with Knuth's Algorithm D (multi-word long division, *TAOCP* Vol. 2 [10]) gives a measured ∼8× speedup of the matcher kernel — the single largest lever, and the reason per-fill latency reaches the single-digit-nanosecond range.

Byte-parity gate. The GPU matcher must produce byte-identical fills (same fills, quantities, trade IDs, ordering) as the CPU oracle. Enforced by millions of randomized fuzz orders plus a KAT corpus, with zero parity failures. A one-byte divergence fails the gate; matching is consensus-critical, so the GPU path ships only on exact agreement.

Host	Fills/sec	Per-fill	Source
GB10 (Blackwell)	104–126 M	6.9 ns	measured, byte-parity gated
M4 Max	∼250 M	4.15 ns	measured, byte-parity gated

Table 4: GPU CLOB matcher fill throughput, byte-identical to the CPU oracle. The matcher is **dependency-latency-bound**: throughput is limited by the data-dependent chain through each fill, not by raw arithmetic or bandwidth. A single GPU sustains ∼100–250 M fills/sec and does *not* reach 10^9; aggregate rates above one billion per second come from N-node market-sharded aggregation (the deployed path) or from a wider Q64.64 representation that shortens the per-fill dependency chain (not built). Settlement is a market-sharded Block-STM layer with measured conflict-detection at 2.2 M fills/sec, CPU and GPU byte-identical.

5.6 What we do not benchmark

Throughput vs latency. The numbers above are throughput. For a single signature verify (`N=1`), CPU is always faster than any GPU; the correct dispatch is to skip the GPU. We do not report "1× verify on GPU is X microseconds" because nobody dispatches that pattern.

Multi-GPU. The Lux substrate is designed to fan out across multiple GPUs, but the headline numbers are single-GPU (M2 Ultra has one Metal device). Multi-GPU numbers will be reported in a future LP-137 revision once the cluster topology stabilizes.

TFHE end-to-end. The TFHE numbers in this paper are for the underlying lattice arithmetic (NTT, poly_mul). End-to-end TFHE program performance (encrypt→evaluate→decrypt with bootstrapping) is reported separately in LP-013.

6 Security Analysis

6.1 Side-channel posture

Every LP-137 algorithm targets constant-time CPU and branchless GPU. Specifically:

- Field arithmetic (Fp/Fr Montgomery REDC) is constant-time on CPU.
- Scalar multiplication on every supported curve uses windowed scalar mul with constant-time table lookup or branchless conditional point selection (`pt_cmov_affine` on Metal, `cmov_*` on CUDA).
- Hash and AEAD primitives have data-independent control flow on CPU; GPU kernels process independent blocks per warp / thread group.
- Threshold-scheme nonce generation uses an explicitly seeded deterministic stream during testing and a CSPRNG (`getrandom(2)` on Linux, `Security.framework` on Darwin) in production.

6.2 GPU memory hygiene

The GPU-residency invariant (§1) requires that secret-state never leaves attested GPU memory during a chain-local hot path. We enforce this in three layers:

1. **Memory model.** Secret buffers are declared in `private` or `threadgroup` address space, not `constant` or `device`, so that the host cannot read them back without an explicit copy.
2. **Static check.** The per-kernel disassembly check (§4) verifies that no host-bound symbol writes a secret buffer onto a host-readable region.
3. **Runtime attestation.** The GPU device must present an attestation envelope (NVNRAS, SEV-SNP, TDX) before the dispatcher will route hot-path work to it. The envelope verification path is in `luxd/cc/attest/`; the C++ side parses only.

6.3 Test-oracle isolation

The attack model has two adversaries:

1. An external attacker who exploits a vulnerability in the production library.
2. An internal mistake (regression, port error) that diverges the production library from the spec.

The vendored test oracle defends against (2). If the CPU canonical and the test oracle came from the same upstream family, a bug in that upstream would propagate through both sides of the byte-equality check; the test would pass while production is broken. Lux first-party CPU plus an independent vendored oracle is the only structure that catches (2) reliably. This is why the substrate keeps the audited upstreams (`blst`, `c-kzg-4844`, `gnark-crypto`, `PQClean`, `mcl`) at the test boundary only.

6.4 Brand neutrality

No public symbol in the C-ABI or in any header file uses the prefix `LUX_`, `LUXFI_`, or `HANZO_`. The algorithm name is the namespace. This serves three purposes:

- Prevents linker-level conflicts when consuming Lux libraries alongside other crypto libraries.

- Eliminates "vendor symbol leak" — the production binary's symbol table cannot be used to fingerprint the substrate.
- Aligns with the formal verification posture: the algorithm's spec is the contract, not Lux's branding.

6.5 Quantum posture

Lux's post-quantum substrate (ML-DSA [29], ML-KEM [28], SLH-DSA [30], FALCON, Pulsar) is documented per-algorithm in its own LP. In aggregate: the consensus pipeline operates in *triple* mode (BLS + Pulsar + ML-DSA), with each layer independently toggleable. A break of any single layer does not break consensus; only the simultaneous compromise of all three does.

Hash-based signatures (SLH-DSA, Lamport [11]) provide a quantum-secure fallback path that depends only on hash preimage resistance. Grover halves effective security, so a 256-bit hash drops to 128-bit quantum security — still more than adequate for current attacker models.

7 Related Work

7.1 Vendor-as-canonical alternative

The most common alternative to LP-137's first-party-CPU-canonical structure is to vendor an audited upstream (e.g. `blst` for BLS12-381, `mcl` for BN254) and use it as the production library. This was specifically considered and rejected on 2026-04-27 (LP-137 `Rejected proposals`). The reasoning:

- **Adversarial-distance loss.** If the production CPU body is the same upstream as the test oracle, the byte-equality check loses its bug-detection power.
- **Vendor symbol leak.** Linking `libblst` into production re-exposes its symbol table and ties the security boundary to upstream's release schedule.
- **Brand fragmentation.** Each vendor uses a different naming convention; merging multiple vendors into a single substrate creates an N-way symbol-prefix conflict.
- **Build fragility.** `aarch64` Apple Silicon is a known weak spot for several upstream BLS libraries; a first-party body removes this dependency.

The verified-perf claim that vendor implementations are "5-10× faster" did not hold up on the cold path (the CPU canonical is not the hot path; the GPU kernel is). On the hot path, the comparison is GPU vs vendor-CPU, not Lux-CPU vs vendor-CPU, so the vendor-CPU faster baseline is irrelevant.

7.2 Comparison with other substrates

`gnark-crypto` (Consensys). Excellent Go reference for pairing-friendly curves and ZK tooling [5]. Lux uses it as a test oracle (§4). It is not a GPU library and does not address the GPU-residency invariant.

`blst` (Supranational). The reference BLS12-381 implementation [33]. CPU-only (with hand-tuned assembly per platform), no GPU kernel. We use it as a test oracle.

`arkworks-rs`. Modular Rust crypto with strong type safety [1]. We import it as a second oracle for BN254 and Banderwagon KAT cross-verification.

`PQClean`. The canonical post-quantum reference repository [31]. ML-KEM [28], ML-DSA [29], SLH-DSA [30] reference implementations are imported via FetchContent; the LP-137 substrate provides the GPU kernel and the C-ABI shim.

`lattigo`. Go reference for FHE primitives (BFV, BGV, CKKS, TFHE) [6]. Lux's TFHE substrate started from this lineage and ships first-party Metal/CUDA NTT kernels for the lattice arithmetic, lifted past the radix-2 ceiling by the LP-164 six-step transform [22, 12].

Rust BLS GPU prototypes (`gnark/golang.org/x/crypto/bls12381_gpu`). Several research-grade GPU BLS prototypes exist. None ship a unified byte-equality contract across CPU + Metal + CUDA + WGSL; LP-137 is, to our knowledge, the first production-grade substrate with that property.

7.3 Why not Conan?

`Conan` was rejected as the C++ dependency manager. Reasons:

- Adds a Python prerequisite to the build pipeline.
- Pulls dependencies from a third-party repo by default; we want every dep to be a Lux fork pinned to a tag.
- Doesn't compose well with CMake's `FetchContent`, which is the standard Lux build idiom.

`FetchContent` is the canonical mechanism. Each forked dep is at `luxcpp/<dep>@<tag>`; the `<alg>/test/cmake/<oracle>.cmake` pulls the test-oracle commit into the test binary only.

8 Conclusion

LP-137 specifies the GPU mode shared by every cryptographic primitive in the Lux 9-chain topology. The substrate is operational on all three backends (Metal, CUDA, WGSL), is byte-equal to the same Go reference on $N = 1000$ random inputs per primitive, and satisfies the GPU-residency invariant: no chain-local hot path leaves attested GPU memory in production.

The per-algorithm formal specs live in their own LPs. LP-137 (this paper) carries no per-algorithm content beyond a pointer index (§3). This separation lets the substrate evolve (new GPU backend, new attestation profile, new dispatcher heuristic) without re-opening per-algorithm specs, and lets each algorithm evolve (new KAT vectors, new constant-time refinement) without re-opening the substrate.

Future work. Three substrate-level extensions are tracked:

1. Multi-GPU dispatcher: extend the determinism harness to verify byte-equality across heterogeneous GPU backends in the same cluster.
2. AMD ROCm and Intel oneAPI backends: the same byte-equality contract applied to two more shader languages, joining Metal/CUDA/WGSL.
3. Hardware-attested GPU memory pinning: integrate with the LP-127 attestation envelope so that the determinism harness can require an attestation receipt for every shipped GPU build.

The 1-LP-per-algorithm decomposition (LP-026, LP-029, LP-066, LP-069, LP-070–077, LP-110–126, LP-131, LP-146–159) is the canonical entry point for any consumer who needs a specific primitive's formal spec. LP-137 is the substrate that holds them together.

References

[1] arkworks contributors. `arkworks`: A Rust ecosystem for zkSNARK programming. https://github.com/arkworks-rs/algebra, 2024.

[2] Vitalik Buterin and Christian Reitwiessner. Eip-196: Precompiled contracts for addition and scalar multiplication on the elliptic curve alt_bn128. https://eips.ethereum.org/EIPS/eip-196, 2017.

[3] Vitalik Buterin and Christian Reitwiessner. Eip-197: Precompiled contracts for optimal ate pairing check on the elliptic curve alt_bn128. https://eips.ethereum.org/EIPS/eip-197, 2017.

[4] Deirdre Connolly, Chelsea Komlo, Ian Goldberg, and Christopher A. Wood. The flexible round-optimized Schnorr threshold (FROST) protocol for two-round Schnorr signatures. RFC 9591, IETF, 2024.

[5] ConsenSys. `gnark-crypto`: A Go library for finite-field arithmetic, elliptic-curve cryptography, and pairings. https://github.com/Consensys/gnark-crypto/releases/tag/v0.19.2, 2026. Version v0.19.2; pinned as test oracle for BN254 and BLS12-381.

[6] EPFL-LDS / Tune Insight. `Lattigo`: A library for lattice-based homomorphic encryption in Go. https://github.com/tuneinsight/lattigo, 2024.

[7] Armando Faz-Hernandez, Sam Scott, Nick Sullivan, Riad S. Wahby, and Christopher A. Wood. Hashing to elliptic curves. RFC 9380, IETF, 2023.

[8] Google. Project wycheproof: cryptographic test vectors. https://github.com/C2SP/wycheproof, 2024.

[9] Simon Josefsson and Ilari Liusvaara. Edwards-curve digital signature algorithm (EdDSA). RFC 8032, IETF, 2017.

[10] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley. Algorithm D (multiple-precision division), §4.3.1.

[11] Leslie Lamport. Constructing digital signatures from a one-way function. SRI Tech Report CSL-98, 1979.

[12] Patrick Longa and Michael Naehrig. Speeding up the number-theoretic transform for faster ideal lattice-based cryptography. In *CANS 2016*, 2016.

[13] Lux Core Team. Lp-124: Ed25519 signature scheme. https://github.com/luxfi/lps/blob/main/LP-124-ed25519.md, 2026.

[14] Lux Core Team. Lp-137 benchmarks: Reproducible bench-ladder json and `ACTUAL-STATE` reconciliation. https://github.com/luxfi/lps/blob/main/LP-137-BENCHMARKS.md, 2026.

[15] Lux Core Team. Lp-137: Gpu-native crypto stack. https://github.com/luxfi/lps/blob/main/LP-137.md, 2026.

[16] Lux Core Team. Lp-146: Bn254 pairing-friendly curve. https://github.com/luxfi/lps/blob/main/LP-146-bn254.md, 2026.

[17] Lux Core Team. Lp-147: secp256k1 curve and ecdsa. https://github.com/luxfi/lps/blob/main/LP-147-secp256k1.md, 2026.

[18] Lux Core Team. Lp-160: Batched montgomery inversion (3-backend uniform). https://github.com/luxfi/lps/blob/main/LP-160-batch-inv.md, 2026.

[19] Lux Core Team. Lp-161: Multi-curve pippenger msm. https://github.com/luxfi/lps/blob/main/LP-161-multi-curve-msm.md, 2026.

[20] Lux Core Team. Lp-162: Combined-pair miller loop. https://github.com/luxfi/lps/blob/main/LP-162-combined-miller.md, 2026.

[21] Lux Core Team. Lp-163: Big-integer karatsuba modexp. https://github.com/luxfi/lps/blob/main/LP-163-karatsuba-modexp.md, 2026.

[22] Lux Core Team. Lp-164: Extended-n lattice ntt (six-step). https://github.com/luxfi/lps/blob/main/LP-164-six-step-ntt.md, 2026.

[23] Lux Core Team. Lp-165: Pedersen tree-reduce vector commit. https://github.com/luxfi/lps/blob/main/LP-165-pedersen-tree-reduce.md, 2026.

[24] Lux Core Team. Lp-166: Threshold pre-signing batch kernel. https://github.com/luxfi/lps/blob/main/LP-166-threshold-presign.md, 2026.

[25] Lux Industries, Inc. Lux lightspeed dex: On-chain order book with sub-second finality. Lux Technical Report.

[26] NIST. FIPS 180-4: Secure hash standard (SHS). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf, 2015.

[27] NIST. FIPS 202: SHA-3 standard: Permutation-based hash and extendable-output functions. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf, 2015.

[28] NIST. FIPS 203: Module-lattice-based key-encapsulation mechanism standard (ML-KEM). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.pdf, 2024.

[29] NIST. FIPS 204: Module-lattice-based digital signature standard (ML-DSA). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.204.pdf, 2024.

[30] NIST. FIPS 205: Stateless hash-based digital signature standard (SLH-DSA). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.205.pdf, 2024.

[31] PQClean contributors. `PQClean`: Clean, portable, tested implementations of post-quantum cryptography. https://github.com/PQClean/PQClean, 2024.

[32] Markku-Juhani Saarinen and Jean-Philippe Aumasson. The BLAKE2 cryptographic hash and message authentication code. RFC 7693, IETF, 2015.

[33] Supranational. `blst`: Multilingual BLS12-381 signature library. https://github.com/supranational/blst, 2024.