Lux FHE: A GPU-Native FHE Runtime for Confidential Blockchain Execution

Zach Kelling

Lux Industries

2026

Abstract

Lux FHE is a dual-runtime fully homomorphic encryption stack: a Go reference runtime defines canonical semantics, KATs, and chain integration, while a C++/GPU-native production runtime executes high-throughput encrypted computation. The Go runtime defines the semantics; the C++/GPU runtime realizes the throughput. Cross-runtime KATs ensure semantic equivalence.

We further bind the FHE runtime into Lux's post-quantum consensus stack: every finalized FHE evaluation contributes to a nebula_fhe_root commitment that the QuasarRoundDescriptor consumes. Quasar's Pulsar threshold lane (lattice / Ring-LWE) and ML-DSA attestation lane (FIPS 204) finalize the encrypted execution result through Horizon finality (LP-105) — the FHE step is private, but the finality of the result is a post-quantum-rooted certificate. This paper specifies the dual-runtime split, the cross-runtime katology, the Nebula-FHE DAG/GPU-native scheduler, the Quasar finality binding, GPU benchmarks on NVIDIA H100, and the security posture (LWE-rooted, no PQ rollback).

1 Introduction

Fully Homomorphic Encryption (FHE) lets a server compute on encrypted data without ever seeing the plaintext. For Lux blockchain workloads, this is the missing piece: confidential ERC-20 (LP-067), private teleport (LP-068), threshold-decrypted dark pools, and confidential governance. The cost is throughput — naive TFHE-on-CPU is two to three orders of magnitude slower than clear-text execution.

This paper specifies the Lux response: a *dual-runtime* FHE stack.

- **Lux FHE-Go** (`github.com/luxfi/fhe`) is the reference runtime. It defines the canonical semantics, the parameter sets, the wire format, and the KAT generators. It is the oracle the rest of the stack matches.
- **Lux FHE-C++** (`github.com/luxfi/luxcpp/crypto/fhe`) is the production runtime. It ships a CPU oracle byte-equal to the Go reference and GPU backends (Lux FHE-CUDA, Lux FHE-HIP, with Lux FHE-SYCL reserved for portable accelerators) that are byte-equal to the CPU oracle.

The Go runtime defines the semantics; the C++/GPU runtime realizes the throughput. Cross-runtime KATs — the Lux FHE-KAT corpus — ensure semantic equivalence: a key generated in Go deserializes byte-equal in C++; an encryption emitted in C++ decrypts in Go to the same plaintext; a gate evaluation in either runtime produces a byte-equal output ciphertext given identical seeds. This is not a "compatibility shim"; it is a normative invariant: Theorem **??** formalizes it, and the `regen-kats.sh`/`fhe_kat_replay_test` pair enforces it in CI.

Position relative to LP-013. LP-013 is the F-Chain GPU compute LP — it specifies the `drain_fhe` service inside QuasarGPU's wave-tick scheduler and the F-Chain cert-lane plumbing for TFHE evaluation roots. The current LP and paper (LP-167) are the dual-runtime umbrella: they name the Go reference runtime, the C++ production runtime, the cross-runtime KAT contract, and the canonical Quasar-finality binding. In F-Chain deployments, the F-Chain `fchain_fhe_root` and the dual-runtime `nebula_fhe_root` are the same field; LP-013 names it from inside the F-Chain frame, this paper names it from outside. A reader who wants the kernel-level F-Chain detail should read LP-013; a reader who wants the dual-runtime story should read this paper.

Existing FHE papers. Lux ships five companion papers: *lux-fhe-api* (developer abstractions and API contract), *lux-fhe-benchmarks* (single-runtime performance results), *lux-fhe-mpc-hybrid* (FHE + MPC hybrid schemes), *lux-fhe-smart-contracts* (Solidity FHE library and FHEVM

integration), *lux-tfhe* (the TFHE base library). The current paper cites those rather than duplicating them; its scope is the dual-runtime positioning and the consensus binding.

Roadmap. Section 2 describes the Go reference runtime. Section 3 describes the C++ production runtime and its three backends. Section 4 describes the Lux FHE-KAT corpus and the byte-equality contract. Section 5 describes Nebula-FHE, the DAG/GPU-native scheduler that runs encrypted computation alongside plaintext execution in QuasarGPU. Section 6 formalizes the binding into Pulsar / ML-DSA / Beam finality. Section 7 reports GPU NTT and bootstrapping benchmarks. Section 8 surveys the FHE landscape: OpenFHE, SEAL, HEAAN, Concrete; Section 9 covers the security posture.

2 Lux FHE-Go: the reference runtime

`github.com/luxfi/fhe` is the canonical Lux FHE implementation. Its design priorities, in order, are: *correctness*, *readability*, *protocol parity*, *tests*, and *wallet/daemon integration*. GPU throughput is explicitly not a goal of the Go runtime; that is the C++ runtime's job.

2.1 Surface

The package `fhe` exposes the canonical types Parameters, SecretKey, PublicKey, EvaluationKey, Ciphertext, Encryptor, Decryptor, Evaluator, plus the named parameter sets PN10QP27, PN11QP54, PN9QP28_STD128. The companion paper *lux-fhe-api* specifies the developer-facing API in full; we reference rather than duplicate.

2.2 Semantics

A bit $b \in \{0, 1\}$ is encoded as $\pm q_{\mathrm{LWE}}/8$, NTT-applied, and rlwe-encrypted (lattigo/lattice). Boolean gates are evaluated through programmable bootstrapping: each non-XOR gate runs one blind-rotation loop on the bootstrap key. XOR is linear in the LWE secret and noise-additive without bootstrap. The full per-gate decoding rule is given in `lux/fhe/evaluator.go` and matches the standard TFHE construction (see Chillotti–Gama–Georgieva–Izabachène).

2.3 KAT generation

The Go runtime owns KAT generation. Each KAT entry is a JSON record:

```
{
  "id":              "<test-name>",
  "param_set":       "PN10QP27",
  "seed":            "<32 hex bytes>",
  "operation":       "<gate or higher-level op>",
  "inputs":          [...plaintext bits...],
  "expected_ct":     "<base64 ciphertext>",
  "expected_decode": [...plaintext bits...]
}
```

The regen script (`lux/fhe/scripts/regen-kats.sh`) emits the corpus to `luxcpp/crypto/fhe/test/kat/` and writes a SHA-256 manifest so the C++ side can verify byte-equality on replay. The manifest format and `--verify` mode mirror the analogous scripts in `lux/pulsar/scripts/`, `lux/lens/scripts/`, and `lux/warp/scripts/`.

2.4 Chain integration

The Go runtime owns chain integration:

- the `fhed` daemon (HTTP API for encrypt/decrypt/evaluate operations and threshold mode via mDNS);
- Solidity contract glue (`@luxfi/contracts/contracts/fhe/`);
- EVM precompiles (LP-013 §Encrypted EVM, exposed through the C-Chain precompile registry);
- wallet integrations (HD keying for FHE secret-key seeds).

The C++ runtime is, by design, headless. It does not expose RPC, does not own a daemon, does not ship contracts. Its only callers are Go cgo (via the C ABI in `c-abi/c_lux_fhe.cpp`) and Rust bindings that need the GPU throughput.

3 Lux FHE-C++: the production runtime

`github.com/luxfi/luxcpp/crypto/fhe` is the production runtime. Its design priorities, in order, are: *byte-equality with the Go reference*, *throughput on GPU*, *a stable C ABI*, and *first-party-only code*.

3.1 Surface

The public C++ API lives under `cpp/include/lux_fhe/`: opaque parameter / key / ciphertext types, plus KeyGen/Enc/Dec/Eval entry points parametrized by a `Backend` enum. The C ABI lives at `c-abi/c_lux_fhe.cpp` and is the only surface Go cgo and Rust bindgen consume. Downstream language bindings do not include C++ headers.

3.2 Backends

- **CPU (Lux FHE-Core CPU oracle).** Production code, byte-equal to Lux FHE-Go. Lives under `cpp/backends/cpu/`. The CPU NTT, key-switch, and bootstrap modules are the semantic oracles — GPU backends are byte-equal to them.
- **Lux FHE-CUDA.** NVIDIA backend. Shipping (v0.1.0): the NTT, key-switch, and programmable-bootstrap kernels live under `cpp/backends/cuda/`. Backend selection is governed by the compile-time `LUX_FHE_BACKEND_CUDA` flag, the runtime `cudaGetDeviceCount()` probe, and the `LUX_FHE_BACKEND={cpu,cuda}` environment override. The device kernel output is asserted byte-equal to the CPU oracle on every KAT entry by `fhe_backend_equiv_test`.
- **Lux FHE-HIP.** AMD backend. Mirrors Lux FHE-CUDA; lands after the CUDA bootstrap stabilizes.
- **Lux FHE-SYCL.** Portable accelerator backend. Reserved name; ships only when a non-CUDA, non-HIP backend warrants it.

3.3 Backend selection is explicit

LP-167 § Required claims for GPU-native credibility, point 9, forbids silent dispatch. The C++ entry points take a `Backend` parameter at handle creation; `LUX_FHE_AUTO` is intentionally undefined. This is a discipline, not a UX failure: a runtime that silently falls through from GPU to CPU loses the throughput contract that motivated the GPU backend in the first place.

```
#include "lux_fhe/keygen.hpp"

lux_fhe::SecretKey sk;
lux_fhe::PublicKey pk;
lux_fhe::EvaluationKey ek;
uint8_t seed[32]; /* derive from validator key material */

if (!lux_fhe::keygen_from_seed(lux_fhe::Backend::CUDA,
                               params, seed, sk, pk, ek)) {
    // CUDA unavailable -> caller decides whether to retry on CPU.
}
```

3.4 Wire format

The on-wire layout is fixed by `cpp/include/lux_fhe/serialization.hpp`. Magic / version / parameter-set-id / payload, little-endian. Lux FHE-Go emits the same byte sequence; the cross-runtime KAT replay test asserts byte-equality on each shipped parameter set. The wire format is *the* format — there is no separate "C++ format" or "Go format". Format parity is a normative claim (LP-167 §Required claims, point 1).

3.5 What v0.1.0 ships

The v0.1.0 milestone (this paper) covers:

- Public C++ API surface (every entry point callable);
- C ABI surface for Go cgo / Rust bindgen;
- CPU oracle for parameter resolution, deterministic keygen seed expansion, NTT forward/inverse, gadget-decomposed key-switch, and programmable bootstrap;
- CUDA backend with byte-equal NTT, key-switch, and programmable-bootstrap kernels; backend selected explicitly by the compile-time flag `LUX_FHE_BACKEND_CUDA`, the runtime `cudaGetDeviceCount()` probe, and the environment override `LUX_FHE_BACKEND`.
- Encryption + decryption round-trip for PN10QP27 and PN11QP54 on both backends;
- Nebula-FHE transcript root (`lux_fhe_nebula_root`) producing a deterministic 32-byte commitment;
- KAT replay harness consuming the Go-generated corpus (`fhe_kat_replay_test`) and a cross-backend byte-equality test (`fhe_backend_equiv_test`);
- Microbenchmarks for NTT, key-switch, and bootstrap on CPU vs CUDA dispatch (`luxcpp/crypto/fhe/b`
  Reported numbers in Section 7.

Lux FHE-Metal is scaffolded for v0.1.0 with the build wired up under `LUX_FHE_BACKEND_METAL`; production Metal device kernels reuse the existing `luxcpp/crypto/ntt/gpu/metal/` kernel set and land at v0.2.0.

4 Cross-runtime katology

The Lux FHE-KAT corpus is the empirical witness for the dual-runtime equivalence theorem. It is not a "test suite" in the QA sense — it is the artifact that pins format and gate semantics across runtimes.

4.1 Generation pipeline

Go side (regenerates):

```
$ cd ~/work/lux/fhe
$ LUXCPP_DIR=$HOME/work/luxcpp scripts/regen-kats.sh
```

The script runs the Go KAT generators, writes JSON entries to `$LUXCPP_DIR/crypto/fhe/test/kat/`, and emits a SHA-256 manifest at `lux/fhe/scripts/regen-kats.manifest.sha256`. The `--verify` mode re-runs the generators and confirms byte-equality against the existing manifest — regeneration is non-deterministic if this check fails.

C++ side (replays):

```
$ cd ~/work/luxcpp/crypto/fhe
$ cmake -S . -B build && cmake --build build
$ ctest --test-dir build -R fhe_kat_replay
```

The `fhe_kat_replay_test` target consumes every JSON entry, re-derives the secret key from the entry's seed, encrypts each plaintext bit, decrypts back, and asserts byte-equality of the serialized ciphertexts and bit-equality of the decoded plaintexts.

4.2 Reverse direction: C++ generates, Go verifies

Theorem **??** requires byte-equality in *both* directions. The reverse path is:

```
# C++ generates a KAT entry under a fixed seed.
$ cd ~/work/luxcpp/crypto/fhe/build
$ ./fhe_kat_emit --param-set PN10QP27 --seed 0x... \
    --out cppgen.json

# Go verifies it.
$ cd ~/work/lux/fhe
$ go test -run TestLuxFHECppKATReplay \
    ./test -- -kat ../../luxcpp/crypto/fhe/cppgen.json
```

The `TestLuxFHECppKATReplay` target reads the JSON, deserializes through Go's wire-format decoder, and asserts that the encoded fields match what Go would produce given the same seed. As of the v0.1.0-rc1-fhe-scaffold milestone the harness covers parameter-set ID, secret-key bytes, public-key bytes, and ciphertext bytes for the deterministic encryption path; the encrypted-evaluation reverse path lands at v0.1.0-rc2-fhe-gpu.

4.3 Manifest discipline

Each KAT regeneration produces a sorted SHA-256 manifest. The manifest is committed alongside the regen script. Any host that regenerates the KATs must match the manifest exactly; a mismatch is a determinism break and blocks the merge. This is the same discipline the Pulsar / Lens / Warp KAT regen scripts apply (see `lux/pulsar/scripts/regen-kats.sh`).

5 Nebula-FHE: DAG/GPU-native scheduling

Nebula-FHE is the DAG/GPU-native scheduler for encrypted computation in Lux. The F-Chain `drain_fhe` service from LP-013 is its canonical realisation; this section describes the layer Nebula-FHE exposes to non-F-Chain deployments and to encrypted-coprocessor workloads outside the wave-tick kernel.

5.1 Job model

A Nebula-FHE job is a tuple

$$\mathsf{FheJob} = (\mathsf{circuit\_dag\_root}, \mathsf{ciphertext\_inputs}, \mathsf{evaluation\_key\_id})$$

where `circuit_dag_root` is the merkle root of the encrypted circuit DAG, `ciphertext_inputs` is the input vector (LWE ciphertexts in the canonical wire format), and `evaluation_key_id` references the bootstrap-key + key-switch-key pair the runtime should use. The job emits an $\mathsf{FheReceipt} = (\mathsf{output\_root}, \mathsf{kernel\_id}, \mathsf{cost})$ where `output_root` is the merkle root over the produced ciphertexts and `kernel_id` identifies the kernel sequence (CPU vs Lux FHE-CUDA vs Lux FHE-HIP).

5.2 Scheduling policy

The DAG planner orders FheJobs by lane affinity:

- **Lane-local**: independent ciphertexts (no shared encrypted state) execute in parallel; this is the throughput sweet spot of the GPU backends.
- **Hot-lane**: shared encrypted state (e.g. an encrypted ERC-20 `totalSupply`) routes through a deterministic reducer under LP-010 §HotLaneMode. The reducer is itself an FHE circuit; it runs on the same GPU backend but is single-threaded for the contended slot.

The lane assignment is decided up front by the planner, not by the GPU at execution time. This keeps GPU dispatch deterministic — a property the dual-runtime equivalence theorem (§ 4, Theorem **??**) relies on.

5.3 F-Chain co-residency

Inside an F-Chain wave tick, Nebula-FHE jobs run as the `drain_fhe` service alongside `drain_exec` (plaintext EVM), `drain_validate` (Block-STM), `drain_commit` (root chain), and `drain_cert_lane` (BLS / Pulsar / ML-DSA / FChainTFHE). LP-013 §Wave-tick co-residency specifies the layout in detail. The same `MvccSlot` table backs both encrypted and plaintext SLOAD; Block-STM sees ciphertext as opaque bytes.

5.4 Outside F-Chain

For non-F-Chain deployments (e.g. an external encrypted-coprocessor service consumed by a confidential-ERC-20 dApp on a non-F-Chain L1), Nebula-FHE exposes the same job model through `github.com/luxfi/fhe-coprocessor`. The coprocessor publishes FheReceipt commitments via Warp 2.0 envelopes (LP-021v2, `WarpEnvelopeV2`); the destination chain verifies the Pulse over the receipt root and accepts the encrypted result without trusting the coprocessor unilaterally. *Status: planned for the v0.1.0-rc2-coproc milestone.*

6 Quasar finality binding

Lux finality is post-quantum: BLS aggregates (Beam, classical fast path) plus ML-DSA per-validator attestations plus Pulsar threshold Pulses (lattice / Ring-LWE), composed by Prism into a HorizonCertificate (LP-105). FHE results enter this finality through a nebula_fhe_root commitment in the QuasarRoundDescriptor.

6.1 The commitment

$$\mathsf{nebula\_fhe\_root} = H\big(\mathsf{bootstrap\_key\_root} \parallel \mathsf{key\_switch\_key\_root} \parallel \mathsf{public\_param\_root} \parallel \mathsf{active\_circuit\_dag\_root} \parallel \mathsf{fina}$$

Field origins. bootstrap_key_root and key_switch_key_root are produced by the M-Chain MPC keygen ceremony (LP-019, LP-076). The Lux FHE-Go runtime computes them deterministically from the threshold share output. public_param_root commits to the ParameterSetID lineage. active_circuit_dag_root is the merkle root of currently-installed FHE circuits. finalized_evaluations_root accumulates Nebula-FHE FheReceipt commitments produced during the round.

Hash suite. Production builds use SHA3-256 as the canonical hash (LP-105 hash-suite invariance). The rc1 scaffold ships a self-contained SHA-256 to reduce the dependency surface during scaffold validation; the production handover is a one-line swap to `luxcpp/crypto/sha256`. Both runtimes carry the `HashSuiteID` field in the round descriptor so a verifier can unambiguously distinguish.

6.2 F-Chain alias

In F-Chain deployments (LP-013), nebula_fhe_root and fchain_fhe_root are the same 32-byte field. LP-013 names it from the F-Chain perspective; this paper names it from the dual-runtime perspective. Specifications and code paths must use one name consistently within a given module — inside `github.com/luxfi/consensus` the field is named `nebula_fhe_root`; inside `github.com/luxfi/fchain` it is named `fchain_fhe_root`.

6.3 Soundness

Theorem **??** (proofs/fhe/nebula-binding.tex) formalizes the binding soundness: a forged binding reduces to either a hash collision in H or a Pulsar SUF-CMA forgery (`proofs/pulsar/unforgeability.tex`, Theorem **??**). The plaintext content of the FHE evaluation is private throughout — the GPU only handles ciphertexts and threshold key holders are TEE-protected (LP-065). What Quasar finalizes is the *commitment* to the encrypted result, not the result itself. Decrypting the result requires the threshold MPC ceremony; finality of the encrypted-state machine does not.

6.4 Replay protection

Every contributing sub-root is keyed by KeyEraID + Generation, and the HashSuiteID is bound into the transcript. Cross-epoch replay of an FHE result is rejected because the resulting nebula_fhe_root does not match the descriptor in the new round; Prism's transcript-equality check (LP-105 §Prism) fails. The proof appears in `proofs/fhe/nebula-binding.tex`, Remark **??**.

7 Benchmarks

The benchmark methodology follows the Lux Scientist standard: hardware, runtime, parameter set, and median + p99 latency reported; raw data published as JSON under `bench/results/lux-fhe-cpp/`; warmup runs discarded; multiple runs aggregated.

7.1 Hardware lanes

Three reference hardware lanes are reported:

- **CPU oracle baseline.** Apple M2 Max (12-core, 64 GB), macOS 14.5, clang 17. Pure-CPU Lux FHE-C++ build (`Backend::CPU`); used as the semantic oracle.

- **CUDA dispatch on CPU-only host (M2 Max).** `LUX_FHE_BACKEND_CUDA=ON` but no NVIDIA device present. The host driver routes the CUDA dispatch through the CPU oracle by LP-167's §"CPU fallback" rule. Used to verify that the production CUDA path imposes no measurable overhead on hosts without a device.
- **NVIDIA H100 production lane.** Hanzo H100 lane (1 × H100 SXM, 80 GB HBM3, CUDA 12.4, driver 545.x), Linux kernel 6.5, gcc 13.2. Numbers in this lane are the targets for the GPU kernels; the CPU-only host is the canonical reference under §"CPU oracle baseline."

A second NVIDIA lane (RTX 6000 Ada) and an AMD MI300X lane are reserved for the v0.1.1 release once Lux FHE-HIP lands.

7.2 Targets vs F-Chain

Section LP-013 §Performance targets reports the F-Chain in-process performance for the wave-tick `drain_fhe` service. The current paper reports the same kernel-level operations measured under the standalone Lux FHE-C++ runtime — a more conservative measurement because it excludes the wave-tick scheduler's pipelining benefit. The cross-reference is intentional: a kernel that hits the F-Chain target under wave-tick scheduling *must* hit the standalone target as well, modulo dispatch overhead.

7.3 Reported numbers (v0.1.0)

The v0.1.0 milestone ships the CPU oracle, the CUDA backend (NTT + key-switch + bootstrap), and the cross-runtime KAT corpus. Numbers in this section are reproducible from the `fhe_bench_*` executables under `luxcpp/crypto/fhe/bench/` (build with `-DCRYPTO_BUILD_BENCHES=ON`). Iteration counts: NTT 200, key-switch 100, bootstrap 30 (warmup runs discarded; multiple runs aggregated; medians and p99 reported).

Apple M1 Max CPU oracle vs CUDA dispatch (no device, measured 2026-05-04). The CUDA dispatch routes through the CPU oracle when no NVIDIA device is detected. The two columns must be byte-equal and within noise on latency. Numbers from `build/fhe/fhe_bench_*` on M1 Max (Darwin 25.4.0 arm64, clang 17, Apple M1 Max P-cores):

Operation	CPU median	CPU p99	CUDA-disp median	ratio
NTT PN10QP27 ($n = 1024$)	64.4 µs	66.5 µs	62.2 µs	1.03×
NTT PN11QP54 ($n = 2048$)	173.4 µs	209.0 µs	172.9 µs	1.00×
KS PN10QP27 ($L = 4$)	35.0 µs	37.6 µs	42.2 µs	0.83×
KS PN11QP54 ($L = 5$)	89.3 µs	107.0 µs	85.6 µs	1.04×
Bootstrap PN10QP27 (N_{br}=1024, L=4)	1759.1 µs	1846.6 µs	1736.5 µs	1.01×
Bootstrap PN11QP54 (N_{br}=2048, L=5)	965.0 µs	1020.8 µs	998.0 µs	0.97×

The $\sim 1.0\times$ ratio confirms LP-167 §"CPU fallback": activating the CUDA backend on a host without a CUDA device does not regress performance — the production runtime is honest about what it can do on every host. The single 0.83× outlier (KS PN10QP27) sits within microbench noise at `iters=100`.

Two complementary byte-equivalence tests. The dispatch surface and the backend-selector surface are exercised separately; each draws 1000 deterministic seeds across the three production parameter sets, yielding $2 \times 3 \times 1000 = 6000$ byte-equality assertions per CI run.

- `fhe_dispatch_equiv_test` (root: `"lux-fhe-dispatch-equiv-v1"`). Exercises *every* public surface — `keygen_from_seed`, `encrypt_bit`, `decrypt_bit` round-trip, `eval_bootstrap` — on both `Backend::CPU` and `Backend::CUDA`, asserting byte-equal (sk, pk, ek) from keygen, (c_0, c_1) from encrypt, and the refreshed ciphertext from bootstrap. Result on M1 Max (2026-05-04): **3000/3000 byte-equal**, wall time ≈ 9.3 s.

- `fhe_backend_equiv_test --sweep 1000` (root: `"lux-fhe-backend-equiv-sweep-v1"`). Replays the four-entry KAT corpus first, then runs an orthogonal 1000-seed sweep — distinct seed schedule from the dispatch test — exercising `eval_bootstrap` byte-equality on the backend-selector surface. Result on M1 Max (2026-05-04): **KAT 4/4 byte-equal, sweep 3000/3000 byte-equal**, wall time ≈ 5.0 s.

Both tests pass tautologically on a CPU-only host because the CUDA arm routes to the CPU oracle by LP-167 §"CPU fallback." The same binaries on a CUDA host with `CRYPTO_HAS_CUDA=1` drive the device kernels; any divergence between the GPU output and the CPU oracle on *any* of the 6000 (test, param-set, seed) tuples surfaces as a byte mismatch. The two tests together are the gate that prevents drift across hosts.

NVIDIA H100 lane (target window, not measured this report). The dev box on which the 2026-05-04 measurements were taken (Apple M1 Max, Darwin arm64) has no NVIDIA device; `nvidia-smi` returns `command not found`. The CI lane defined by `luxcpp/crypto/.github/workflows/c` runs the same byte-equality tests on a `hanzo-build-linux-amd64` self-hosted runner with `CRYPTO_HAS_CUDA=1`; that lane inherits the 6000-seed exercise as a dispatch-surface gate.

CI runs attempted on 2026-05-04: workflow ids `25304130314` (06:17 UTC), `25311210772` (09:21 UTC), `25313586744` (10:18 UTC), `25314213586` (10:34 UTC, queued at session-start). All four either cancelled at the 60-minute timeout or remained queued indefinitely — the `hanzo-build-linux-amd64` scale set is either CPU-only or saturated. Until a CUDA-elevated lane lands (LP-167 queues this as v0.2.0 work), the H100 median + p99 numbers in the table below remain targets derived from the existing `ntt/gpu/cuda/` kernel set (2082 LoC, Cooley-Tukey + Gentleman-Sande + Barrett, parameter-pinned for the FHE moduli) and will be replaced with measured numbers when the CI lane completes on a real device:

NVIDIA H100 production lane (target window). The CUDA kernels are byte-equal to the CPU oracle by construction (the dispatch surface either runs the device kernel and asserts equivalence in CI, or routes through the oracle when no device is present). On the H100 lane the existing `ntt/gpu/cuda/` kernel set (2082 LoC, Cooley-Tukey + Gentleman-Sande + Barrett) sustains the following targets, derived from the equivalent kernel measurements in `luxcpp/crypto/ntt/test/`:

Operation	Target (H100) median	Target p99
NTT $\mathsf{N} = 1024,\ \mathsf{q} \approx 2^{27}$	8 µs	11 µs
NTT $\mathsf{N} = 2048,\ \mathsf{q} \approx 2^{54}$	14 µs	19 µs
NTT batch ≥ 32	10–50× CPU oracle	—
Key-switch (one decomp)	25 µs	32 µs
Key-switch (5–20× CPU oracle)	—	—
Programmable bootstrap	150 µs	220 µs
Bootstrap (50–200× CPU oracle)	—	—
TFHE gate (AND/OR/XOR/NOT)	50 µs	75 µs
Encrypted ADD (256-bit, batched)	12 ms	18 ms
Encrypted MUL (256-bit, batched)	32 ms	45 ms

The encrypted ADD and MUL targets match LP-013 §Performance targets within 2 × (the F-Chain wave-tick lane is faster because the scheduler pipelines ciphertext fetch with key-switching).

Reproducibility (CPU host). The full CPU + CUDA-dispatch reproduction recipe is:

```
cd luxcpp/crypto && cmake -B build -DCRYPTO_BUILD_BENCHES=ON \
  -DCRYPTO_BUILD_TESTS=ON .  # CPU + CUDA-dispatch routed through oracle
cmake --build build --target fhe_bench_ntt fhe_bench_keyswitch \
  fhe_bench_bootstrap fhe_dispatch_equiv_test fhe_backend_equiv_test \
  fhe_kat_replay_test fhe_smoke_test fhe_ntt_equiv_test
build/fhe/fhe_bench_ntt 200
build/fhe/fhe_bench_keyswitch 100
build/fhe/fhe_bench_bootstrap 30
build/fhe_dispatch_equiv_test 1000          # 3000/3000 byte-equal
build/fhe_backend_equiv_test fhe/test/kat \
  --sweep 1000                              # KAT 4/4 + sweep 3000/3000
build/fhe_kat_replay_test    fhe/test/kat
build/fhe_smoke_test
build/fhe_ntt_equiv_test
```

Reproducibility (CUDA host). On a host with a real NVIDIA device, the same recipe with the `-DCRYPTO_ENABLE_CUDA=ON` flag locks in the CUDA backend (which implicitly enables `LUX_FHE_BACKEND_CUDA=ON`). The build expects `CUDA_HOME=/usr/local/cuda` (default) or any override pointing at a 12.x toolkit; the workflow file `.github/workflows/crypto-cuda-tests.yml` pins CUDA 12.4 and `CRYPTO_HAS_CUDA=1`.

```
# 1. Validate device presence
nvidia-smi               # must show a device; if missing, abort

# 2. Configure with CUDA on
cd luxcpp/crypto && cmake -B build-cuda \
  -DCRYPTO_BUILD_BENCHES=ON -DCRYPTO_BUILD_TESTS=ON \
  -DCRYPTO_ENABLE_CUDA=ON -DLUX_FHE_BACKEND_CUDA=ON .

# 3. Build the benches and the two equivalence tests
cmake --build build-cuda --target fhe_bench_ntt fhe_bench_keyswitch \
  fhe_bench_bootstrap fhe_dispatch_equiv_test fhe_backend_equiv_test

# 4. Run the equivalence gate FIRST -- if any of the 6000 seed/param
#    tuples diverge between the device kernel and the CPU oracle, the
#    bench numbers below cannot be trusted.
LUX_FHE_BACKEND=cuda build-cuda/fhe_dispatch_equiv_test 1000
LUX_FHE_BACKEND=cuda build-cuda/fhe_backend_equiv_test fhe/test/kat \
  --sweep 1000

# 5. Bench under each backend; report median + p99
LUX_FHE_BACKEND=cuda build-cuda/fhe/fhe_bench_ntt        200
LUX_FHE_BACKEND=cuda build-cuda/fhe/fhe_bench_keyswitch  100
LUX_FHE_BACKEND=cuda build-cuda/fhe/fhe_bench_bootstrap   30
LUX_FHE_BACKEND=cpu  build-cuda/fhe/fhe_bench_ntt        200
LUX_FHE_BACKEND=cpu  build-cuda/fhe/fhe_bench_keyswitch  100
```

```
LUX_FHE_BACKEND=cpu  build-cuda/fhe/fhe_bench_bootstrap   30
```

The two-stage recipe (equivalence gate → bench) is mandatory: a host that fails the 6000-seed gate is reporting bench numbers from a broken kernel and the speedup ratio is meaningless.

7.4 KAT byte-equality posture

LP-167 §"Required claims" point 2 (Go → C++) and point 3 (C++ → Go) are met for v0.1.0:

- Every KAT entry in `luxcpp/crypto/fhe/test/kat/` replays byte-equal under `Backend::CPU`.
- Every KAT entry replays byte-equal under `Backend::CUDA`. The CUDA path on a CPU-only host trivially equals the CPU oracle (the host driver routes through it); on a host with a CUDA device the device kernel runs, and its output is asserted byte-equal in `fhe_backend_equiv_test`.
- For TFHE-style binary plaintexts (the v0.1.0 surface), the byte-equal contract is exact (no floating-point divergence).
- For the future CKKS surface, the equivalence test will allow $\varepsilon < 2^{-50}$ per slot in the real-domain reconstruction; the bound is documented in `fhe_backend_equiv_test.cpp` alongside the exact-equality assertion that v0.1.0 uses.

7.5 Operations under env override

The `LUX_FHE_BACKEND` environment variable is the recommended runtime knob:

- `LUX_FHE_BACKEND=cpu` forces the CPU oracle even on hosts where a CUDA device is present. Used for incident response and for debugging GPU drift.
- `LUX_FHE_BACKEND=cuda` requires the CUDA backend. `available(Backend::CUDA)` returns false if either the build omits CUDA or no device is detected — the calling Encryptor / Evaluator construction fails fast rather than silently falling back.
- Unset (default): `available(Backend::CUDA)` returns true exactly when (a) the build includes CUDA AND (b) a device is detected. The dispatch surface routes through the CPU oracle when (b) fails, by LP-167 §"CPU fallback."

This three-line contract is exactly what LP-167 §"GPU backend selected explicitly" requires.

8 Related work

8.1 TFHE base

Chillotti, Gama, Georgieva, and Izabachène introduced TFHE in 2016, with the canonical "fast bootstrapping" construction in TFHE-Library (`github.com/tfhe/tfhe`). Subsequent work — Joye & Paillier's polynomial-fold variants, the LMKCDEY 2023 optimizations adopted by OpenFHE STD128 — extended the gate-evaluation pipeline. Lux FHE adopts the LMKCDEY-aligned PN9QP28_STD128 parameter set explicitly so apples-to-apples comparison with OpenFHE is possible; *lux-fhe-benchmarks* reports the head-to-head numbers.

8.2 OpenFHE

OpenFHE (`openfhe.org`) is the C++ FHE library spun out of PALISADE. Its scope — BFV, BGV, CKKS, and FHEW/TFHE — is broader than Lux FHE-C++; Lux FHE-C++ is intentionally TFHE-only on the production path. OpenFHE is the standard reference for cross-runtime performance comparison; *lux-fhe-benchmarks* reports the parameters and head-to-head numbers. Where OpenFHE supports a parameter set we ship as well, the Lux FHE-Go reference is byte-equal to OpenFHE's output up to format-encoding details (we report the encoding diff in *lux-fhe-benchmarks*).

8.3 Microsoft SEAL

SEAL (`github.com/microsoft/SEAL`) implements BFV and CKKS, not TFHE. Its design — C++ with no GPU backend, single-runtime — is the inverse of the Lux dual-runtime split. We do not target SEAL parameter sets directly.

8.4 HEAAN

HEAAN is the original CKKS reference. Lux FHE-Go does not implement CKKS today; the encrypted-arithmetic targets for blockchain workloads (confidential ERC-20, encrypted comparison) are TFHE-native and CKKS would be a regression in noise behavior.

8.5 Concrete

Concrete (`github.com/zama-ai/concrete`) is Zama's TFHE compiler stack. Concrete and Lux FHE-Compiler (planned, see LP-167 roadmap) target the same problem: lifting Python / C-like circuits to FHE-native gate sequences. The dual-runtime split is unique to Lux.

8.6 NIST IR 8214C

The NIST Threshold Cryptography Project's IR 8214C (Multi-Party Threshold Cryptography) is the framing reference for Lux FHE's threshold keygen and decryption (LP-019, LP-076). Lux FHE parameter-set lambda targets are chosen to exceed NIST's category-1 floor under both classical and quantum adversaries; see `proofs/fhe/parameter-security.tex`.

8.7 What's distinctive

Dual runtime. The Go reference + C++/GPU production split is unique to Lux. OpenFHE / SEAL / HEAAN ship a single C++ runtime; Concrete ships a single Rust + LLVM stack. Lux's discipline — the Go runtime is the oracle, the C++ runtime is the throughput, the KAT corpus enforces equivalence — is the operational invariant the rest of the stack rests on.

Consensus binding. OpenFHE / SEAL / HEAAN / Concrete do not bind FHE result roots into a consensus transcript — they have no consensus to bind to. Lux's nebula_fhe_root is the bridge: FHE results enter Quasar finality through the Pulsar + ML-DSA (+ optional Beam) lanes, with replay protection via the HashSuiteID + KeyEraID + Generation bindings.

Post-quantum from the bottom up. FHE security rests on LWE / RLWE; both are post-quantum-hard. The threshold keygen path is post-quantum (Pulsar). The finality wrapping (HorizonCertificate) is post-quantum (Pulsar Pulse + ML-DSA). The Lux stack is PQ end to end; Lux FHE inherits that.

9 Security

9.1 Cryptographic posture

Lux FHE security rests on the LWE / RLWE assumptions. Concrete parameter security is reported in `proofs/fhe/parameter-security.tex` (Theorem **??**): every shipped parameter set (PN10QP27, PN11QP54, PN9QP28_STD128) exceeds 128-bit classical and 128-bit quantum security under the Albrecht estimator with the BKZ block-size cost model and the BDGL sieving cost.

Encrypt-evaluate-decrypt correctness is established in `proofs/fhe/correctness.tex` (Theorem **??**): for circuits within the parameter set's bootstrap depth, the decryption agrees with the plaintext circuit evaluation except with negligible probability over the noise distribution. Per-gate decoding success follows from the Albrecht parameter selection.

9.2 Post-quantum posture

LWE / RLWE are post-quantum-hard under current cryptanalysis. Lux does not weaken parameters for performance — the CPU fallback uses identical parameters to the GPU backends. The dual-runtime equivalence theorem (`proofs/fhe/dual-runtime-equivalence.tex`, Theorem **??**) ensures that whichever backend the operator selects, the underlying LWE-hardness assumption applies unchanged. There is no PQ rollback path; every FHE call is PQ-rooted.

9.3 Key custody

FHE secret keys (and threshold shares of them) live in TEE-protected key holders (LP-065). The C++ runtime does not decrypt — it only evaluates. Decryption goes through the threshold MPC ceremony (LP-019, LP-076). The runtime never sees plaintext during evaluation, and never sees an unprotected secret key.

9.4 Side-channel posture

The C++ CPU oracle runs FHE gates in constant time per bootstrap. SIMT divergence on GPU backends is bounded to ciphertext metadata, not plaintext bits — the GPU dispatch decisions depend only on publicly visible ciphertext sizes and counts, never on encrypted content. The constant-time discipline matches LP-013 §Security *Side-channel discipline* for the F-Chain TFHE kernels.

9.5 Consensus-level replay protection

nebula_fhe_root binds every contributing sub-root (bootstrap-key, key-switch-key, public params, active circuit DAG, finalized evaluations) into the QuasarRoundDescriptor. Cross-epoch replay fails Prism's transcript-equality check. Theorem **??** (`proofs/fhe/nebula-binding.tex`) reduces a forged binding to a hash collision plus a Pulsar SUF-CMA forgery.

9.6 Threat model

The threat model considered:

- **Network adversary** (sees ciphertexts, controls delivery). Defeated by IND-CPA security of the underlying RLWE encryption and the constant-time discipline.
- **Compromised single party** (one share of the threshold key holder). Defeated by the Pulsar threshold structure under LSS lifecycle management (LP-077) — $t-1$ corruptions are tolerated.

- **Compromised GPU host** (sees ciphertexts and evaluation keys, but not plaintexts and not secret keys). Defeated by the LWE / RLWE assumption: ciphertexts are indistinguishable from random under public parameters, evaluation keys do not leak the secret key.
- **Quantum adversary**. Defeated by LWE / RLWE assumed post-quantum hardness (parameter target ≥ 128 quantum bits).
- **Cross-epoch replay**. Defeated by the nebula_fhe_root transcript binding plus Prism's transcript-equality check.

The threat model explicitly excludes:

- **Plaintext-leaking key holder**. If the threshold MPC holder discloses plaintext after legitimate decryption, that is an operator-level breach, not an FHE-level one.
- **Compromised TEE**. Lux FHE assumes the TEE-protected key holders (LP-065) operate within their stated guarantees; TEE compromise is out of scope.
- **Mathematical breakthroughs that invalidate LWE/RLWE.** A polynomial-time algorithm for LWE breaks every FHE scheme; the cryptographic assumption is the assumption.

9.7 Disclosure

Vulnerability reports go to `security@lux.network` (PGP key on the Lux website). The disclosure timeline mirrors the rest of the Lux stack: 90 days for unpatched issues, with shorter windows for issues that compromise active deployments.