FHE API and Developer Model for Practical Private Applications

Zach Kelling

Lux Industries

2025

Abstract

We describe the Go API and developer model for Lux FHE, an original lattice-based fully homomorphic encryption implementation built on `luxfi/lattice`. The library provides boolean gate evaluation on encrypted bits with programmable bootstrapping, radix-decomposed encrypted integers (uint8–uint256), and SIMD-optimized NTT for polynomial arithmetic. This paper specifies the developer-facing abstractions: parameter selection, key lifecycle (generation, serialization, threshold distribution), encryption/decryption patterns, evaluator design, error handling, batching strategies for amortized cost, and safe-by-default API patterns that prevent common misuse (key reuse, nonce collision, ciphertext malleability). The API is designed for blockchain integration via EVM precompiles and the `fhed` daemon's HTTP interface, with supported circuits including comparison, arithmetic, bitwise operations, and conditional selection.

1 Introduction

FHE enables computation on encrypted data without decryption, but its practical adoption is limited by two problems: performance and usability. Performance is addressed in the companion benchmarking paper [1]. This paper addresses usability: how should a developer interact with an FHE library without needing to understand lattice cryptography, noise budgets, or bootstrapping schedules?

Lux FHE (`github.com/luxfi/fhe`) is a pure Go implementation built on the `luxfi/lattice/v7` library. It provides:

- Boolean circuit evaluation with automatic bootstrapping after each gate.
- Radix-decomposed integers supporting arithmetic, comparison, and bitwise operations.
- NTT SIMD optimization with Barrett and Montgomery reduction.
- C FFI bindings (`bindings/cabi/`) and Rust bindings (`bindings/rust/`) for cross-language use.
- HTTP API via the `fhed` daemon for integration with non-Go applications.
- Threshold mode with LSSS key sharing and mDNS-based peer discovery.

2 Parameter Selection

2.1 Standard Parameter Sets

Developers choose from three pre-validated parameter sets:

Name	Security	LWE Dim	Use Case
`PN10QP27`	128-bit classical	$N = 1024$	Default, good performance
`PN11QP54`	128-bit classical	$N = 2048$	Higher precision
`PN9QP28_STD128`	128-bit classical	$N = 512/1024$	OpenFHE compatible

Table 1: Standard parameter sets. All use NTT-friendly primes $Q \equiv 1 \pmod{2N}$.

Custom parameter construction is deliberately not exposed in the public API. Choosing incorrect parameters can silently degrade security. Developers who need non-standard parameters must use the internal `ParametersLiteral` type and accept responsibility for security analysis.

2.2 Parameter Encoding

Parameters define the LWE dimension, blind rotation dimension, modulus, and base-two decomposition level. The encoding uses the `luxfi/lattice/v7` RLWE parameter structure:

```
params, err := fhe.NewParameters(fhe.PN10QP27)
if err != nil {
    return fmt.Errorf("create params: %w", err)
}
// params.NLWE() = 1024
// params.NBR()  = 1024
// params.QLWE() = 0x7fff801
```

3 Key Lifecycle

3.1 Generation

Key generation produces three artifacts: a secret key, a public evaluation key (bootstrap key), and optionally a key-switching key:

```
sk, bsk, err := fhe.GenerateKeys(params)
if err != nil {
    return fmt.Errorf("keygen: %w", err)
}
enc := fhe.NewEncryptor(params, sk)
eval := fhe.NewEvaluator(params, bsk)
dec := fhe.NewDecryptor(params, sk)
```

The evaluator does not require the secret key. This separation is the fundamental safety property: the entity performing computation never sees plaintext.

3.2 Serialization

All key types implement `encoding.BinaryMarshaler` and `encoding.BinaryUnmarshaler`:

```
data, err := sk.MarshalBinary()
// store securely via KMS

var sk2 fhe.SecretKey
err = sk2.UnmarshalBinary(data)
```

3.3 Threshold Key Distribution

For threshold mode (t-of-n), key generation uses LSSS (Linear Secret Sharing Scheme):

```
shares, bsk, err := fhe.GenerateThresholdKeys(
    params, t, n,
)
// Distribute shares[i] to party i
// bsk (bootstrap key) is public
```

Resharing to a new (t', n') configuration does not require reconstructing the original secret key:

```
newShares, err := fhe.Reshare(
    shares, newT, newN,
)
```

4 Encryption and Decryption

4.1 Boolean Encryption

Single-bit encryption produces an LWE ciphertext:

```
ct, err := enc.EncryptBit(true)
bit, err := dec.DecryptBit(ct)
// bit == true
```

4.2 Integer Encryption

Integers are decomposed into radix blocks, each block being a `ShortInt` (1–4 bits) encrypted in a single LWE ciphertext:

```
intEnc := fhe.NewIntegerEncryptor(params, sk)
ct, err := intEnc.EncryptUint32(42)
val, err := intEnc.DecryptUint32(ct)
// val == 42
```

The `RadixCiphertext` type carries metadata about its FHE type (`FheUint8`, `FheUint16`, `FheUint32`, `FheUint64`, `FheUint128`, `FheUint256`) and enforces type matching at the API boundary.

5 Evaluator Operations

5.1 Boolean Gates

The evaluator supports standard boolean gates, each followed by automatic bootstrapping:

Gate	Method
AND, OR, XOR, NOT	`eval.AND(a, b)`, etc.
NAND, NOR, XNOR	`eval.NAND(a, b)`, etc.
MUX (ternary select)	`eval.MUX(cond, a, b)`
MAJORITY (2-of-3)	`eval.MAJORITY(a, b, c)`

Table 2: Boolean gate API. Each returns `(*Ciphertext, error)`.

5.2 Integer Operations

The `IntegerEvaluator` provides arithmetic and comparison on radix-decomposed encrypted integers:

Category	Operations
Arithmetic	Add, Sub, Mul, Div, Rem, Neg
Comparison	Eq, Ne, Lt, Le, Gt, Ge
Bitwise	And, Or, Xor, Not, Shl, Shr
Selection	Mux (conditional select)
Scalar	AddScalar, MulScalar, SubScalar

Table 3: Integer evaluator operations. All accept `*RadixCiphertext` arguments.

5.3 Error Handling

Every operation returns an explicit error. Common error conditions:

- **Type mismatch**: Attempting to add a `FheUint8` to a `FheUint32`. Rejected at the API boundary.
- **Nil ciphertext**: Passing an uninitialized ciphertext. Returns `ErrNilCiphertext`.
- **Parameter mismatch**: Evaluating with keys generated for different parameters. Returns `ErrParamMismatch`.

No operation panics. All errors are returned, never swallowed.

6 Batching and Amortization

Boolean gates have high per-gate cost due to bootstrapping. Batching amortizes the cost of key material transfer and NTT setup across multiple operations.

The `fhed` daemon's `/evaluate` endpoint accepts batched operations:

```
// Batch of 100 AND gates in a single request
results, err := client.EvaluateBatch(
    fhe.OpAND, pairs,
)
```

Batched evaluation reuses the NTT engine's precomputed twiddle factors and Montgomery constants across all operations in the batch, reducing amortized setup cost to near zero.

7 Safe-by-Default Patterns

The API enforces several safety invariants:

1. **No key reuse across parameter sets**: Attempting to use a key generated for `PN10QP27` with `PN11QP54` parameters returns an error at construction time, not at operation time.
2. **Ciphertext type tagging**: Every ciphertext carries its FHE type. Operations between mismatched types are rejected before any computation occurs.
3. **No plaintext in evaluator**: The `Evaluator` type does not accept or produce plaintext values. The only path to plaintext is through the `Decryptor`, which requires the secret key.

4. **Deterministic randomness for testing**: The `fhe.NewDeterministicEncryptor()` variant uses a seeded PRNG for reproducible tests but is marked as unsafe for production via build tags.

8 EVM Integration

FHE operations are exposed as EVM precompiles on Lux C-Chain:

Precompile	**FHE Operation**
FHEAdd, FHESub, FHEMul	Encrypted integer arithmetic
FHEEq, FHELt, FHEGt, FHELe, FHEGe	Encrypted comparison
FHEAnd, FHEOr, FHEXor, FHENot	Encrypted bitwise
TrivialEncrypt	Plaintext $\rightarrow$ ciphertext (public value)
Decrypt, Reencrypt	Async decryption via T-Chain

Table 4: FHE EVM precompiles at addresses `0x0200...0080`–`0x0200...0083`.

Solidity developers interact via the `@luxfi/contracts` library:

```
import "@luxfi/contracts/FHE.sol";
euint32 a = FHE.asEuint32(encrypted_input);
euint32 b = FHE.add(a, FHE.asEuint32(10));
```

9 Conclusion

FHE is unusable if the API leaks implementation complexity to the developer. Lux FHE's API provides three levels of abstraction: boolean gates for circuit designers, integer operations for application developers, and EVM precompiles for Solidity developers. All levels enforce type safety, prevent key misuse, and return explicit errors. The threshold mode extends the same API to distributed settings without changing the programming model. Performance characteristics of these operations are detailed in the companion benchmarking paper [1].

References

[1] Kelling, Z. (2025). *FHE Benchmarking, Cost Models, and Operational SLOs*. Lux Industries Research.

[2] Kelling, Z. (2025). *Threshold Fully Homomorphic Encryption for Confidential DeFi*. Lux Industries Research.

[3] Kelling, Z. (2025). *Quasar: Quantum-Secure Multi-Engine Consensus with Triple-Proof Quantum Finality*. Lux Industries Research.

[4] Kelling, Z. (2024). *Threshold Decryption and Recovery for Confidential Systems*. Lux Industries Research.

[5] Kelling, Z. (2025). *Secure Enclave, HSM, and Threshold Boundary Design*. Lux Industries Research.

[6] Chillotti, I., Gama, N., Georgieva, M., & Izabachene, M. (2020). *TFHE: Fast Fully Homomorphic Encryption over the Torus*. Journal of Cryptology, 33(1), 34–91.

[7] Lee, Y., Micciancio, D., Kim, A., Choi, R., Deryabin, M., Eom, J., & Yoo, D. (2023). *Efficient FHEW Bootstrapping with Small Evaluation Keys*. EUROCRYPT 2023.