Fully Homomorphic Encryption for Smart Contract Execution

Zach Kelling

Lux Industries

2024

Abstract

We present an architecture for executing smart contracts on fully homomorphic encrypted (FHE) data, enabling private DeFi—decentralized finance where transaction amounts, balances, and orderbook entries are encrypted on-chain and manipulated without decryption. The implementation uses Ring-LWE-based FHE with native Go and NTT SIMD acceleration (not tfhe-rs, not Zama), providing a self-contained cryptographic backend. We describe the FHE virtual machine architecture deployed on the LiquidFHE chain, including encrypted state transitions, noise budget management, and bootstrapping strategies for long-running computations. Practical applications include encrypted orderbooks (dark pools compliant with Regulation ATS), private governance voting, and sealed-bid auctions. We provide performance benchmarks: bootstrapping at 12 ms per gate (TFHE boolean), 45 ms for 16-bit encrypted addition (BFV batched), and 180 ms for 16-bit encrypted multiplication. The FHE precompile is deployed at address `0x0700000000000000000000000000000000000000` on the Lux C-Chain, with gas costs calibrated to computational overhead.

Keywords: fully homomorphic encryption, FHE, smart contracts, private DeFi, Ring-LWE, encrypted computation, dark pools

Contents

1 Introduction **4**

2 FHE Primer **4**
2.1 Ring Learning with Errors . . . 4
2.2 Encryption and Homomorphic Operations . . . 4
2.3 Bootstrapping . . . 5
2.4 Noise Management . . . 5

3 Lux FHE Implementation **5**
3.1 Design Principles . . . 5
3.2 NTT Acceleration . . . 6

4 Encrypted State Transitions **6**
4.1 State Model . . . 6
4.2 Encrypted ERC-20 Token . . . 6
4.3 Verification of Encrypted Transitions . . . 7

5 FHE VM Architecture on LiquidFHE Chain **7**
5.1 Chain Configuration . . . 7
5.2 Precompile Interface . . . 7
5.3 Decryption Authorization . . . 7

6 Practical Applications **8**
6.1 Encrypted Orderbook (Dark Pool) . . . 8
6.2 Private Governance Voting . . . 8
6.3 Sealed-Bid Auctions . . . 9

7 Performance **9**
7.1 TFHE Boolean Operations . . . 9
7.2 BFV Batched Arithmetic . . . 9
7.3 Comparison to Cleartext EVM Execution . . . 9

8 Security Analysis **10**
8.1 Cryptographic Security . . . 10
8.2 Semantic Security . . . 10

9 Related Work **11**

10 Conclusion **11**

1 Introduction

Blockchain transparency is simultaneously its greatest feature and its most significant limitation. Every transaction amount, every account balance, every smart contract state variable is publicly visible. This transparency enables trustless verification but prohibits use cases that require confidentiality:

- **Dark pools**: Institutional trading requires hidden orderbooks. On transparent chains, every pending order is visible, enabling front-running.
- **Private voting**: On-chain governance with visible votes before tallying enables vote-buying and last-minute strategic voting.
- **Sealed-bid auctions**: Auction mechanisms (Vickrey, Dutch) require bid secrecy until the reveal phase. Transparent chains leak bids.
- **Regulatory compliance**: Regulation ATS (Alternative Trading Systems) requires that certain trading data remain confidential from non-participants.

Fully homomorphic encryption (FHE) resolves this tension by enabling computation on encrypted data. A smart contract operating on FHE ciphertexts can add encrypted balances, compare encrypted values, and update encrypted state—all without decrypting. Only authorized parties (key holders) can decrypt the final results.

This paper describes the Lux FHE implementation: a native Go cryptographic backend with NTT SIMD acceleration, integrated as an EVM precompile and deployed on the LiquidFHE chain.

2 FHE Primer

2.1 Ring Learning with Errors

FHE security rests on the hardness of the Ring Learning with Errors (Ring-LWE) problem [2].

Definition 2.1 (Ring-LWE)**.** *Let $R_q = \mathbb{Z}_q[X]/(X^N+1)$ be a polynomial ring with N a power of 2. The Ring-LWE problem: given polynomials $(a_i, b_i = a_i \cdot s + e_i)$ where $s \in R_q$ is a secret and e_i are drawn from a discrete Gaussian distribution χ with small standard deviation σ, distinguish (a_i, b_i) from uniform random pairs.*

No known quantum algorithm provides more than a polynomial speedup over classical algorithms for Ring-LWE. NIST's post-quantum standards (ML-KEM, ML-DSA) are built on the related Module-LWE assumption, providing strong evidence for the security of Ring-LWE-based FHE.

2.2 Encryption and Homomorphic Operations

An FHE scheme encrypts a plaintext m as a ciphertext $\mathsf{ct} = (c_0, c_1)$ where $c_0 = a \cdot s + e + \Delta \cdot m$ and $c_1 = a$, with $\Delta = \lfloor q/t \rfloor$ the scaling factor and t the plaintext modulus.

Homomorphic addition is component-wise:

$$\mathsf{ct}_{\text{add}} = \mathsf{ct}_1 + \mathsf{ct}_2 = (c_{0,1} + c_{0,2}, c_{1,1} + c_{1,2})$$

Decrypting ct_{add} yields $m_1 + m_2$ because the noise terms add linearly: $e_{\text{add}} = e_1 + e_2$.

Homomorphic multiplication is a tensor product followed by relinearization:

$$\mathsf{ct}_{\text{mul}} = \mathsf{Relin}(\mathsf{ct}_1 \otimes \mathsf{ct}_2)$$

Multiplication increases noise quadratically: $\|e_{\text{mul}}\| \approx \|e_1\| \cdot \|e_2\|$. After sufficiently many multiplications, noise exceeds the decryption threshold, and the ciphertext becomes unrecoverable.

2.3 Bootstrapping

Bootstrapping [1] "refreshes" a ciphertext by homomorphically evaluating the decryption function, producing a new ciphertext of the same plaintext with reduced noise. This enables unlimited computation depth at the cost of significant computational overhead.

Definition 2.2 (Bootstrapping)**.** *Given a ciphertext* ct *with noise level* $\|e\|$ *near the threshold, and a bootstrapping key* bsk *(encrypted secret key), bootstrapping produces* $\mathsf{ct}' = \mathsf{Bootstrap}(\mathsf{ct}, \mathsf{bsk})$ *such that* $\mathsf{Dec}(\mathsf{ct}') = \mathsf{Dec}(\mathsf{ct})$ *and* $\|e'\| \ll \|e\|$*.*

Our implementation uses the CGGI bootstrapping technique [3] (programmable bootstrapping via blind rotation), which simultaneously refreshes noise and evaluates a lookup table, enabling gate-level computation in a single bootstrapping pass.

2.4 Noise Management

The central challenge of practical FHE is noise management. Each operation increases noise; bootstrapping resets noise but is expensive. The FHE VM must track noise budgets for every ciphertext and bootstrap before decryption fails.

Table 1: Noise growth by operation type.

Operation	**Noise Growth**	**Depth Cost**
Addition	$\|e_1\| + \|e_2\|$ (linear)	+0
Plaintext multiplication	$c \cdot \|e_1\|$ (linear)	+0
Ciphertext multiplication	$\|e_1\| \cdot \|e_2\|$ (quadratic)	+1
Bootstrapping	$\to e_{\text{fresh}}$ (reset)	0 (reset)

3 Lux FHE Implementation

3.1 Design Principles

1. **Native Go**: The entire FHE backend is implemented in Go using the `luxfi/lattice` library. No Rust FFI, no C bindings, no external dependencies (not tfhe-rs, not Zama's fhevm).

2. **NTT SIMD**: Polynomial multiplication uses the Number Theoretic Transform with AVX-512 SIMD intrinsics via Go assembly. NTT reduces $O(N^2)$ polynomial multiplication to $O(N \log N)$.

3. **Deterministic gas**: Every FHE operation has a fixed gas cost proportional to its computational overhead, enabling predictable transaction pricing.

4. **Scheme selection**: We support two FHE schemes:

   - **TFHE** (Torus FHE): Boolean circuits. 12 ms per bootstrapped gate. Best for comparisons, conditionals, bit-level operations.
   - **BFV** (Brakerski-Fan-Vercauteren): Batched integer arithmetic. 45 ms for 16-bit addition. Best for arithmetic-heavy computations (balance updates, aggregations).

3.2 NTT Acceleration

The Number Theoretic Transform is the computational bottleneck of lattice-based FHE. For polynomials in $R_q = \mathbb{Z}_q[X]/(X^N + 1)$ with $N = 4096$ (BFV) or $N = 1024$ (TFHE), NTT transforms a polynomial from coefficient representation to evaluation representation in $O(N \log N)$ modular multiplications.

Our SIMD implementation processes 8 modular multiplications in parallel using AVX-512 `vpmuludq` instructions. For $N = 4096$:

Table 2: NTT performance (single core, AMD EPYC 7763).

Ring Size N	**Scalar NTT**	**SIMD NTT**
1,024	18 μs	3.2 μs
2,048	42 μs	7.1 μs
4,096	98 μs	15.4 μs
8,192	225 μs	34 μs
16,384	520 μs	78 μs

The SIMD speedup is approximately 5.6–6.7×, consistent with the theoretical maximum of 8× for 8-wide SIMD with memory access overhead.

4 Encrypted State Transitions

4.1 State Model

In the FHE VM, contract state variables are stored as ciphertexts. A contract's storage maps `bytes32` keys to FHE ciphertexts:

$$\mathsf{State} : \texttt{bytes32} \to \mathsf{FHECiphertext}$$

An encrypted state transition T transforms encrypted state S to encrypted state S':

$$S' = T(S, \mathsf{ct}_{\text{input}})$$

where $\mathsf{ct}_{\text{input}}$ is the encrypted transaction input. The transition T is a composition of homomorphic operations (addition, multiplication, comparison) applied to encrypted values. At no point during execution does any plaintext value appear in the EVM state, logs, or transaction receipt.

4.2 Encrypted ERC-20 Token

As a concrete example, consider an encrypted ERC-20 token where balances are FHE ciphertexts:

Algorithm 1 Encrypted ERC-20 transfer (pseudocode).

Require: Sender s, receiver r, encrypted amount ct_{amt}
Require: Encrypted balances ct_s, ct_r
1: $\mathsf{ct}_{\text{sufficient}} \leftarrow \mathsf{FHE.GTE}(\mathsf{ct}_s, \mathsf{ct}_{\text{amt}})$ ▷ Encrypted comparison
2: $\mathsf{ct}'_s \leftarrow \mathsf{FHE.CMux}(\mathsf{ct}_{\text{sufficient}}, \mathsf{FHE.Sub}(\mathsf{ct}_s, \mathsf{ct}_{\text{amt}}), \mathsf{ct}_s)$
3: $\mathsf{ct}'_r \leftarrow \mathsf{FHE.CMux}(\mathsf{ct}_{\text{sufficient}}, \mathsf{FHE.Add}(\mathsf{ct}_r, \mathsf{ct}_{\text{amt}}), \mathsf{ct}_r)$
4: Store ct'_s and ct'_r

The $\mathsf{FHE.CMux}$ (conditional multiplexer) selects between two encrypted values based on an encrypted condition bit, implementing the balance check without revealing whether the sender has sufficient funds.

4.3 Verification of Encrypted Transitions

A challenge unique to FHE smart contracts: validators cannot verify the correctness of encrypted state transitions by re-executing on plaintext (they don't have the plaintext). Instead, validators verify:

1. **Ciphertext well-formedness**: Every input ciphertext has noise within the valid range and was encrypted under the contract's public key.
2. **Operation correctness**: Each homomorphic operation was applied correctly (deterministic given the ciphertexts and operation type).
3. **Noise budget**: The output ciphertext's noise level does not exceed the decryption threshold.

Since FHE operations are deterministic given the same inputs, validators can re-execute the homomorphic computation on the ciphertexts and verify that the output matches. This does not require knowledge of the plaintext.

5 FHE VM Architecture on LiquidFHE Chain

5.1 Chain Configuration

The LiquidFHE chain is a Lux subnet with the following configuration:

Table 3: LiquidFHE chain parameters.

Parameter	**Value**
Block time	2 s
Block gas limit	100M gas
FHE precompile address	`0x0700...0000`
TFHE parameter set	$N = 1024$, $q \approx 2^{27}$, 128-bit security
BFV parameter set	$N = 4096$, $q = 2^{218}$, 128-bit security
Max ciphertext size (BFV)	32 KB
Max ciphertext size (TFHE)	2 KB
Bootstrapping key size	48 MB

5.2 Precompile Interface

The FHE precompile at `0x0700000000000000000000000000000000000000` exposes the following operations via ABI-encoded function selectors:

5.3 Decryption Authorization

Decryption is the most sensitive operation. It must be authorized by the ciphertext owner (the entity whose public key was used for encryption). The FHE precompile enforces access control:

1. Each ciphertext is tagged with an *owner address*.
2. `fheDecrypt` requires a signature from the owner address authorizing decryption of the specific ciphertext handle.
3. Decryption produces a plaintext that is returned to the caller but *not stored on-chain*. The on-chain state remains encrypted.

Table 4: FHE precompile operations and gas costs.

Operation	Selector	Gas Cost
`fheAdd(ct, ct)`	`0x01`	65,000
`fheSub(ct, ct)`	`0x02`	65,000
`fheMul(ct, ct)`	`0x03`	320,000
`fheGte(ct, ct)`	`0x10`	480,000
`fheLte(ct, ct)`	`0x11`	480,000
`fheEq(ct, ct)`	`0x12`	480,000
`fheCMux(ct, ct, ct)`	`0x20`	520,000
`fheBootstrap(ct)`	`0x30`	800,000
`fheEncrypt(pt, pk)`	`0x40`	120,000
`fheDecrypt(ct, proof)`	`0x41`	150,000

6 Practical Applications

6.1 Encrypted Orderbook (Dark Pool)

A dark pool is a trading venue where orders are not visible to other participants until matched. On a transparent blockchain, orderbook entries are publicly visible, enabling front-running and sandwich attacks. FHE eliminates this:

1. **Order submission**: Trader encrypts $(price, quantity, side)$ under the matching engine's public key and submits the ciphertext on-chain.
2. **Matching**: The smart contract performs encrypted comparisons: $\mathsf{FHE.GTE}(\mathsf{ct}_{\text{bid}}, \mathsf{ct}_{\text{ask}})$. Matched orders produce an encrypted execution price.
3. **Settlement**: Only matched counterparties receive decryption keys for their specific trades. Unmatched orders remain encrypted and invisible.

This construction satisfies Regulation ATS requirements for fair access and confidential order handling, as the matching engine operates on encrypted data without ever observing plaintext order flow.

6.2 Private Governance Voting

On-chain governance typically uses transparent voting, which enables:

- Vote buying (verifiable votes can be sold)
- Strategic last-minute voting (seeing the tally before voting)
- Social pressure (publicly visible individual votes)

FHE voting:

1. Each voter encrypts their vote (0 or 1) under the governance contract's public key.
2. The contract homomorphically adds all encrypted votes: $\mathsf{ct}_{\text{tally}} = \sum_i \mathsf{ct}_{\text{vote}_i}$.
3. At the end of the voting period, a threshold decryption (requiring t-of-n governance committee members) reveals only the final tally, not individual votes.

6.3 Sealed-Bid Auctions

Sealed-bid auctions (first-price, second-price/Vickrey) require that bids remain secret until the reveal phase. FHE enables this without a trusted auctioneer:

1. Bidders encrypt their bids under the auction contract's public key.
2. The contract computes $\mathsf{FHE.Max}(\mathsf{ct}_1, \ldots, \mathsf{ct}_n)$ using a sorting network of encrypted comparisons.
3. For Vickrey auctions, the contract computes the second-highest bid homomorphically.
4. The winner is notified via threshold decryption of a per-bidder indicator ciphertext.

7 Performance

All benchmarks on AMD EPYC 7763, single core, Go 1.22 with AVX-512 SIMD NTT.

7.1 TFHE Boolean Operations

Table 5: TFHE gate evaluation benchmarks ($N = 1024$, 128-bit security).

Operation	**Latency**
AND / OR / XOR / NAND / NOR / XNOR	12 ms
NOT	0.02 ms
MUX (conditional select)	24 ms
8-bit addition	310 ms
8-bit comparison	280 ms
16-bit addition	680 ms
16-bit comparison	620 ms

7.2 BFV Batched Arithmetic

Table 6: BFV batched arithmetic benchmarks ($N = 4096$, 128-bit security).

Operation	**16-bit**	**32-bit**
Encrypted addition	45 ms	52 ms
Encrypted subtraction	45 ms	52 ms
Encrypted multiplication	180 ms	340 ms
Plaintext-ciphertext multiply	22 ms	28 ms
Relinearization	85 ms	120 ms
Bootstrapping	1.2 s	2.4 s

7.3 Comparison to Cleartext EVM Execution

The overhead is large in absolute terms (10^6–$10^8\times$). This is inherent to FHE and consistent across all implementations. The relevant comparison is not FHE vs. cleartext, but FHE vs. the *alternative for achieving the same privacy guarantee*:

Table 7: FHE overhead relative to cleartext EVM operations.

Operation	Cleartext	FHE (BFV)	Overhead
`uint256 add`	3 gas (5 ns)	65K gas (45 ms)	9×10^6
`uint256 mul`	5 gas (8 ns)	320K gas (180 ms)	2.3×10^7
`>=` comparison	3 gas (5 ns)	480K gas (480 ms)	9.6×10^7

- Off-chain computation with on-chain verification (ZK proofs): requires a trusted prover and does not support arbitrary computation.
- Trusted execution environments (TEEs): requires trusting hardware vendors and is vulnerable to side-channel attacks.
- Multi-party computation (MPC): requires interaction between parties for every operation and does not scale to public smart contracts.

FHE is the only approach that enables *non-interactive* computation on encrypted data by *untrusted* parties. The overhead is the price of this unique property.

8 Security Analysis

8.1 Cryptographic Security

Both TFHE and BFV derive their security from Ring-LWE [2]. Our parameter selection:

Table 8: Security parameters.

Scheme	N	$\log_2 q$	Classical Security
TFHE	1,024	27	128 bits
BFV-128	4,096	218	128 bits
BFV-192	8,192	438	192 bits
BFV-256	16,384	881	256 bits

Security levels are estimated using the lattice-estimator [6] with BKZ-β cost model and core-SVP hardness assumption.

8.2 Semantic Security

FHE provides IND-CPA (indistinguishable under chosen plaintext attack) security: an adversary cannot distinguish encryptions of different plaintexts. This means:

- Encrypted balances reveal nothing about the plaintext balance.
- Encrypted order prices reveal nothing about the plaintext price.
- Encrypted votes reveal nothing about the plaintext vote.

FHE does *not* provide IND-CCA2 security (indistinguishable under adaptive chosen ciphertext attack). Ciphertexts are malleable by design—this is what enables homomorphic computation. Access control for decryption (Section 5) is therefore critical: only authorized parties can decrypt, preventing an adversary from using the decryption oracle to break semantic security.

9 Related Work

Zama's fhEVM [7] provides FHE for EVM using the TFHE-rs Rust library. Our approach differs in three respects: (1) native Go implementation avoiding FFI overhead, (2) support for both TFHE and BFV scheme selection, and (3) deployment as a subnet with FHE-optimized gas scheduling rather than an L2.

Sunscreen [8] provides a compiler from high-level programs to BFV circuits. Our FHE VM operates at a lower level, exposing individual homomorphic operations as precompile calls, giving smart contract developers fine-grained control over noise budget management.

Inco Network [9] uses Zama's TFHE-rs with a modular blockchain architecture. Like Zama's fhEVM, it relies on Rust FFI. Our native Go implementation eliminates cross-language boundaries and integrates directly with the Lux VM architecture.

10 Conclusion

Fully homomorphic encryption enables a class of smart contract applications—private DeFi, encrypted orderbooks, sealed-bid auctions, private voting—that are impossible on transparent blockchains. The computational overhead (10^6–$10^8\times$ relative to cleartext) is substantial but acceptable for applications where the alternative is not "fast cleartext computation" but rather "no computation at all" (because privacy requirements preclude on-chain execution).

The Lux FHE implementation provides a self-contained, native Go cryptographic backend with SIMD acceleration, deployed as an EVM precompile on the LiquidFHE chain. Bootstrapping at 12 ms per gate (TFHE) and 1.2 s (BFV) enables practical encrypted smart contracts for applications with moderate computational depth.

References

[1] C. Gentry, "Fully Homomorphic Encryption Using Ideal Lattices," *STOC '09*, pp. 169–178, ACM, 2009.

[2] V. Lyubashevsky, C. Peikert, and O. Regev, "On Ideal Lattices and Learning with Errors over Rings," *EUROCRYPT 2010*, pp. 1–23, Springer, 2010.

[3] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast Fully Homomorphic Encryption over the Torus," *Journal of Cryptology*, vol. 33, pp. 34–91, 2020.

[4] Z. Brakerski, "Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP," *CRYPTO 2012*, pp. 868–886, Springer, 2012.

[5] J. Fan and F. Vercauteren, "Somewhat Practical Fully Homomorphic Encryption," IACR ePrint 2012/144, 2012.

[6] M. R. Albrecht, R. Player, and S. Scott, "On the Concrete Hardness of Learning with Errors," *Journal of Mathematical Cryptology*, vol. 9, no. 3, pp. 169–203, 2015.

[7] Zama, "fhEVM: Confidential Smart Contracts on the EVM Using Fully Homomorphic Encryption," Zama Whitepaper, 2023.

[8] Sunscreen, "Sunscreen: A Compiler for FHE Applications," Sunscreen Tech Report, 2023.

[9] Inco Network, "Inco: Modular Confidential Computing Network," Inco Whitepaper, 2024.

[10] NIST, "Module-Lattice-Based Key-Encapsulation Mechanism Standard (FIPS 203)," National Institute of Standards and Technology, August 2024.

[11] NIST, "Module-Lattice-Based Digital Signature Standard (FIPS 204)," National Institute of Standards and Technology, August 2024.

[12] O. Regev, "On Lattices, Learning with Errors, Random Linear Codes, and Cryptography," *STOC '05*, pp. 84–93, ACM, 2005.