FHE DARK POOLS

Encrypted Order Matching with
Threshold Decryption and Compliance

Zach Kelling

Satschel, Inc.

Version 1.0 — January 2025

Contents

Abstract

Dark pools account for over half of US equity trading volume, yet existing implementations rely on trusted operators who observe all order flow — creating information leakage, front-running risk, and regulatory opacity. We present the first dark pool protocol built natively on a fully homomorphic encryption (FHE) blockchain, where orders remain encrypted throughout the matching process. Our system implements periodic batch auctions over TFHE-encrypted limit orders on Lux Network's Threshold VM, introducing CMPCOMBINE — a single-bootstrap comparison propagation gate that reduces matching circuit depth by 60%. Settlement occurs through threshold decryption among a committee of validators, with a dedicated regulatory key share enabling SEC Reg ATS compliance without compromising trader privacy. We demonstrate that a batch of 100 encrypted orders can be matched in under 30 seconds on commodity GPU hardware (8×H100), achieving practical throughput for institutional dark pool trading. Our approach eliminates the trusted operator assumption entirely: no party — including the matching engine, validators, or regulators — observes plaintext order data unless the cryptographic threshold is met.

Introduction

The Dark Pool Problem

Dark pools — private trading venues that do not display orders to the public before execution — have grown to dominate equity market microstructure. As of January 2025, dark pools handle approximately 51.8% of US equity volume [1], with over 60 registered Alternative Trading Systems (ATSs) operating under SEC Rule 303 of Regulation ATS. The appeal is straightforward: institutional traders executing large block orders need protection from adverse selection and information leakage that occurs on lit exchanges.

Yet the fundamental architecture of dark pools is paradoxical. Traders seek privacy from the market, but must fully reveal their orders to the dark pool operator. This creates a single point of trust failure:

- **Information leakage.** Operators observe all order flow. Even without malicious intent, data breaches, employee misconduct, and operational errors expose sensitive trading intentions. Between 2014 and 2024, the SEC levied over $170M in fines against dark pool operators for mishandling subscriber order information [2].
- **Front-running and internalization.** Operators or their affiliates may trade against the order flow they observe. The SEC's enforcement actions against Barclays LX and Credit Suisse CrossFinder demonstrated systemic abuse of the trusted operator model [3].
- **Regulatory opacity.** Regulators require post-trade transparency (FINRA Rule 4552, MiFID II post-trade reporting) but have limited tools to verify that matching occurred fairly without observing the matching process itself.

The crypto dark pool problem is even more acute. On-chain order books are fully transparent — every pending transaction in the mempool is visible, enabling MEV extraction. Existing solutions rely on off-chain matching with on-chain settlement, reintroducing the trusted operator assumption, or use zero-knowledge proofs that prove execution correctness but cannot perform computation on encrypted data.

Why Fully Homomorphic Encryption

Fully homomorphic encryption (FHE) enables arbitrary computation on encrypted data without decryption. This capability directly addresses the dark pool trust problem: orders can be encrypted by traders, matched by a smart contract operating entirely on ciphertexts, and decrypted only at settlement by a threshold committee. No single party ever observes plaintext orders.

Prior work has explored MPC-based dark pools (Renegade [4]), ZK-based privacy (Penumbra [5]), and homomorphic commitment schemes. FHE offers a distinct advantage: the matching logic runs as a deterministic program over ciphertexts, verifiable on-chain, without interactive communication rounds between participants during matching. This eliminates the $O(n^2)$ communication complexity of MPC approaches for n-party matching.

However, FHE has been impractical for on-chain use due to performance constraints. Recent advances in TFHE (Torus FHE) [6], GPU acceleration [7], and purpose-built blockchain integration [8] have changed this calculus. TFHE-rs v0.4 achieves 64-bit addition in 186ms on CPU and approximately 8.7ms on an 8×H100 GPU cluster — fast enough for batch processing of institutional-size order books.

Our Contributions

We make the following contributions:

1. **First FHE-native dark pool protocol.** We present a complete dark pool design where order matching executes entirely over TFHE-encrypted data on a blockchain virtual machine, with no plaintext exposure during the matching phase.
2. **CMPCOMBINE optimization.** We introduce a novel circuit optimization for comparison-dominated matching circuits that propagates comparison results through a single programmable bootstrap, reducing matching circuit depth by approximately 60% compared to naive composition.
3. **Practical batch auction design.** We demonstrate periodic batch auctions that match 100 encrypted limit orders in under 30 seconds on GPU, with concrete benchmarks derived from TFHE-rs v0.4 performance data.
4. **Regulatory compliance architecture.** We design a threshold key share model where a regulatory key share enables SEC Reg ATS compliance — regulators can participate in threshold decryption of specific matched trades for audit purposes, without gaining the ability to decrypt arbitrary encrypted orders unilaterally.
5. **Quantified performance model.** We provide a detailed performance model with per-operation benchmarks, throughput analysis, and comparison against existing systems (Renegade, Penumbra, Sunscreen, Zama fhEVM).

Background

TFHE: Torus Fully Homomorphic Encryption

TFHE (Torus FHE) [6] is a fully homomorphic encryption scheme based on the Learning With Errors (LWE) problem over the real torus $\mathbb{T} = \mathbb{R}/\mathbb{Z}$. We summarize the key properties relevant to our construction.

Encryption. A plaintext message $m \in \{0, 1\}$ is encoded on the torus and encrypted as:

```
TLWE_s(m) = (a, b = <a,s> + m*(1/2) + e)
```

where $s \in \mathbb{B}^n$ is the secret key, $a \leftarrow_\$ \mathbb{T}^n$ is uniform random, and e is a small Gaussian error. For integer arithmetic, messages are scaled to $\Delta \cdot m$ where $\Delta = 2^{64}/p$ for plaintext modulus p.

Homomorphic operations. TFHE supports:

- *Addition/Subtraction:* TLWE ciphertexts add component-wise. Cost: negligible (no bootstrap required).
- *Programmable bootstrapping (PBS):* The core operation. Evaluates an arbitrary function $f : \mathbb{Z}_p \to \mathbb{Z}_p$ on an encrypted value while refreshing noise. A single PBS takes ~10ms on CPU for standard parameters.
- *Multiplication:* Implemented via PBS-based decomposition. Requires multiple bootstraps.
- *Comparison:* Implemented as a PBS evaluating the sign function.

Noise management. Each homomorphic operation increases ciphertext noise. When noise exceeds a threshold, decryption fails. PBS simultaneously evaluates a function and resets noise to a fixed level, enabling unbounded computation depth at the cost of one PBS per "noisy" gate.

Security parameter. We use $n = 1024$, $\log_2(q) = 64$, $\sigma = 2^{-25}$ for 128-bit security, following the Homomorphic Encryption Standard [9].

Threshold FHE

Standard FHE requires a single secret key holder. For a dark pool, this is unacceptable — it recreates the trusted operator. Threshold FHE distributes the secret key among t-of-n parties such that:

- Any t parties can collaboratively decrypt
- Fewer than t parties learn nothing about the plaintext

We use the threshold decryption scheme of Boneh et al. [10], adapted for TLWE:

1. **Key generation.** A trusted dealer (or DKG protocol) generates secret key s and distributes shares $s_1, \ldots, s_n$ using Shamir's (t, n) secret sharing over $\mathbb{Z}_q$, such that $s = \sum_i \lambda_i \cdot s_i$ for appropriate Lagrange coefficients λ_i.
2. **Partial decryption.** Given ciphertext $c = (a, b)$, party i computes partial decryption $d_i = b - \langle a, s_i \rangle$.
3. **Reconstruction.** Given t partial decryptions, the plaintext is recovered as $m = \sum_i \lambda_i \cdot d_i \pmod q$, then rounded to the nearest multiple of Δ.

For our construction, we use LSSS (Linear Secret Sharing Scheme) resharing to enable dynamic committee membership without regenerating the master key.

Dark Pool Mechanisms

A dark pool implements a matching engine that pairs buy and sell orders without pre-trade transparency. The two primary matching mechanisms are:

Continuous matching. Orders are matched immediately upon arrival if a counterparty exists. This is the standard model for most existing dark pools (e.g., Barclays LX, Credit Suisse Cross-

Finder). Advantages: low latency. Disadvantages: arrival-order dependence creates gaming opportunities.

Periodic batch auctions. Orders accumulate during a collection period and are matched simultaneously at a uniform clearing price at the end of each period. This is the mechanism used by IEX's D-Limit orders and proposed by Budish, Cramton, and Shim [11] as a solution to high-frequency trading arms races. Advantages: eliminates speed advantages, maximizes matched volume. Disadvantages: higher latency (equal to the batch period).

We adopt periodic batch auctions for two reasons: (1) FHE operations are too expensive for continuous per-order matching, and (2) batch processing amortizes the cost of threshold decryption across all matched trades in a batch.

Clearing price computation. Given a set of encrypted limit orders, the clearing price p^* is the price that maximizes matched volume:

```
p* = argmax_p min( Sum_{buy orders with price >= p} quantity_i,
                   Sum_{sell orders with price <= p} quantity_j )
```

This computation involves comparisons (price $\geq p$), conditional sums, and a max operation — all of which are expressible as TFHE circuits.

Regulatory Context

Dark pools in the US are regulated as Alternative Trading Systems under SEC Regulation ATS (Rules 300–303). Key requirements include:

- **Fair Access (Rule 301(b)(5)):** ATSs with >5% volume in any NMS security must provide fair access.
- **Order Protection (Reg NMS Rule 611):** ATSs must avoid trade-throughs of protected quotations.
- **Post-trade reporting (FINRA Rule 4552):** ATSs must report aggregate trading information.
- **Record-keeping (Rule 302):** ATSs must maintain records of subscribers, orders, and executions for examination by the SEC.

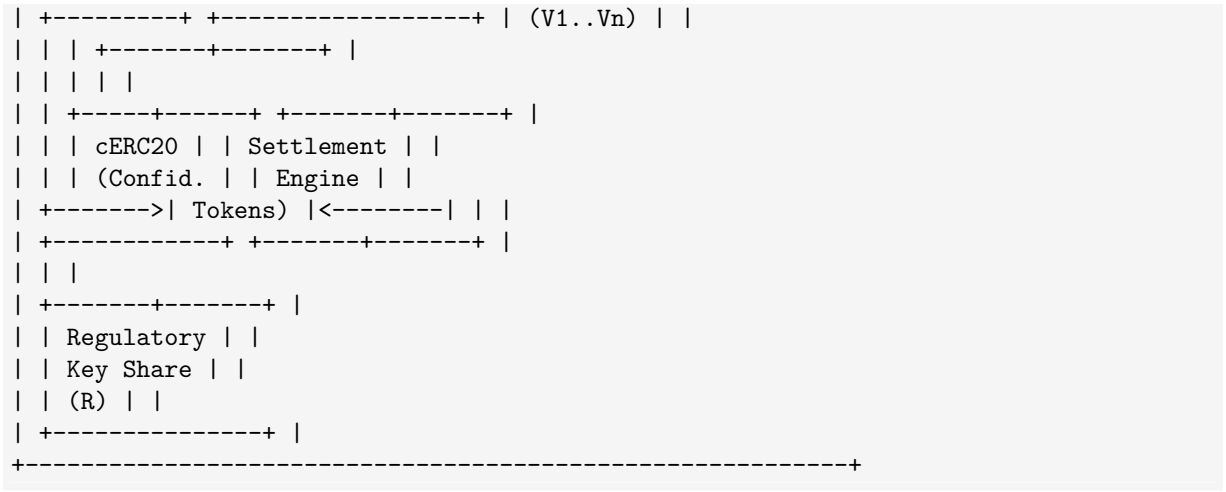
In December 2025, the SEC issued a Request for Information on crypto ATS trading, signaling openness to blockchain-based trading venues provided they meet existing regulatory frameworks [12]. Our threshold key share model is designed to satisfy Rule 302 record-keeping requirements while preserving pre-trade privacy.

System Model and Threat Model

System Actors

Our system involves five classes of participants:

```
+-----------------------------------------------------------+
| Lux Network |
| |
| +---------+ +------------------+ +---------------+ |
| | Traders |-->| DarkPool.sol |-->| Threshold | |
| | (T1..Tm)| | (FHE Matching) | | Committee | |
```



1. **Traders ($T_1, \ldots, T_m$):** Submit encrypted limit orders to the DarkPool contract. Each trader holds their own TFHE encryption key for local order encryption, then re-encrypts under the network public key before submission.

2. **Threshold VM Validators ($V_1, \ldots, V_n$):** Lux validators who hold shares of the FHE decryption key. They execute FHE computations on encrypted orders and participate in threshold decryption at settlement. We require a (t, n) threshold with $t = \lceil 2n/3 \rceil + 1$.

3. **DarkPool Contract:** A smart contract on Threshold VM that implements the batch auction matching logic entirely over encrypted data using FHE precompiles.

4. **Confidential ERC20 (cERC20):** Token contracts where balances and transfer amounts are TFHE-encrypted. Settlement transfers cERC20 tokens between matched traders.

5. **Regulatory Key Share Holder (R):** A designated party (e.g., SEC-registered examiner, FINRA, or compliance officer) who holds one key share in the threshold scheme. R's participation is required for decryption of matched trade records for regulatory examination, but R alone cannot decrypt any data.

Security Goals

We define four security properties:

G1: Pre-trade privacy. No party (including validators, other traders, and the regulator) learns any information about the content of an unmatched order (price, quantity, side, or trader identity) beyond what is implied by the order's existence.

G2: Execution integrity. The matching outcome is correct: the clearing price maximizes volume, all matched orders are filled at the clearing price, and no phantom trades are created. This is verified by the on-chain FHE computation.

G3: Post-trade confidentiality. After matching, only the matched counterparties and (upon regulatory request with threshold participation) the regulator learn the details of matched trades. Unmatched orders remain encrypted.

G4: Regulatory auditability. The regulator, by contributing their key share to a threshold decryption ceremony with $t-1$ validators, can decrypt the matched trade record for any specific batch. The regulator cannot unilaterally decrypt, preventing mass surveillance.

Adversary Model

We consider a computationally bounded adversary $\mathcal{A}$ who may:

- **Corrupt up to $t-1$ validators.** A coalition of fewer than t validators learns nothing about encrypted orders or matched trades. This is the standard threshold assumption.
- **Corrupt traders.** A corrupted trader learns only information about their own orders and matched trades. They cannot extract information about other traders' encrypted orders from the contract state.
- **Observe the blockchain.** $\mathcal{A}$ can see all encrypted ciphertexts, transaction metadata (timing, gas usage, sender address), and contract state. We do not hide the existence or timing of order submissions — only their content.
- **Perform chosen-plaintext attacks.** $\mathcal{A}$ can submit their own orders with known plaintext and observe the resulting ciphertexts. Security follows from the IND-CPA property of TLWE.

Exclusions. We do not protect against:

- Traffic analysis (number of orders per batch is observable)
- Timing side channels on FHE computation (mitigated by fixed batch periods)
- Compromise of t or more validators simultaneously
- Bugs in the FHE implementation (we assume correctness of TFHE-rs)

Protocol Design

Encrypted Order Submission

A trader T submits a limit order by encrypting its components under the network's collective TFHE public key pk:

```
Order = {
    side: Enc_pk(s) where s in {0, 1} (0=buy, 1=sell)
    price: Enc_pk(p) where p in [0, 2^64) (scaled fixed-point)
    quantity: Enc_pk(q) where q in [0, 2^64) (token units)
    owner: Enc_pk(addr) where addr in [0, 2^160)
    nonce: Enc_pk(r) where r <-$ [0, 2^64) (replay protection)
}
```

Each field is encrypted independently as a TFHE ciphertext. The trader also submits a ZK proof π demonstrating:

1. The plaintext price p is within the valid price range $[p_{\min}, p_{\max}]$.
2. The plaintext quantity q is positive and within the trader's deposited collateral.
3. The nonce r has not been used before (checked against a Merkle tree of used nonces).

This proof is generated using the Lux zk-FHE bridge: the trader proves statements about the plaintext before encryption, without revealing the plaintext to the verifier. The proof binds to the ciphertext via a commitment to the randomness used in encryption.

Collateral deposit. Before submitting orders, traders deposit cERC20 tokens into the DarkPool contract as collateral. The contract verifies (homomorphically) that order quantity does not exceed deposited collateral. This prevents unbacked orders without revealing collateral amounts.

Batch Auction Matching on Encrypted Data

Each batch auction proceeds in three phases: collection, matching, and settlement.

Phase 1: Collection (duration: τ_{collect}). Orders accumulate in the contract. At the end of the collection period, the contract freezes the order set $O = \{o_1, \ldots, o_k\}$ for matching.

Phase 2: Matching (duration: τ_{match}). The contract executes the matching circuit on encrypted data using FHE precompiles. The algorithm proceeds as follows:

Price Discovery

We discretize the price space into N price levels $\{P_1, \ldots, P_N\}$ (e.g., $N = 1000$ tick levels). For each price level P_j, we compute the aggregate supply and demand:

```
D(Pj) = Sum_i Select(side_i = BUY AND price_i >= Pj, quantity_i, 0)
S(Pj) = Sum_i Select(side_i = SELL AND price_i <= Pj, quantity_i, 0)
```

where $\text{Select}(\text{cond}, a, b)$ returns a if cond is true (encrypted), b otherwise, implemented as a TFHE CMUX gate.

The clearing price P^* is the price level maximizing matched volume:

```
V(Pj) = Min(D(Pj), S(Pj))
P* = argmax_j V(Pj)
```

The argmax is computed by iterating through price levels and maintaining an encrypted running maximum:

```
best_volume = Enc(0)
best_price = Enc(P1)

for j = 1 to N:
    is_better = V(Pj) > best_volume // FHE comparison
    best_volume = Select(is_better, V(Pj), best_volume)
    best_price = Select(is_better, Pj, best_price)
```

CMPCOMBINE Optimization

The matching circuit is comparison-dominated: price comparisons ($\text{price}_i \geq P_j$), volume comparisons ($V(P_j) >$ best_volume), and min/max operations all require comparison gates. In standard TFHE, each comparison produces an encrypted boolean that must be bootstrapped before use in subsequent operations. Chaining comparisons — e.g., comparing a result that itself depends on a comparison — requires sequential bootstraps, creating a deep circuit.

We introduce CMPCOMBINE, a compound gate implemented in the Threshold VM evaluator that fuses a comparison with its downstream conditional select into a single programmable bootstrap. The key insight is that the PBS lookup table can encode both the comparison and the conditional selection simultaneously.

Standard approach (2 bootstraps):

```
// Bootstrap 1: comparison
flag = PBS(a - b, LUT_sign) // flag = Enc(a >= b ? 1 : 0)

// Bootstrap 2: conditional select
result = PBS(flag * (x - y) + y, LUT_id) // result = Enc(flag ? x : y)
```

CMPCOMBINE (1 bootstrap):

```
// Single bootstrap: comparison + propagation
result = PBS(a - b, LUT_cmpcombine)
// where LUT_cmpcombine encodes: if (a-b) >= 0 then x else y
```

The LUT_cmpcombine table is constructed by composing the sign function with the select function. Since PBS evaluates an arbitrary lookup table in constant time, this fusion is exact — no approximation or precision loss.

Applicability. CMPCOMBINE applies whenever a comparison result is used exactly once as a selector. In the matching circuit, this pattern appears in:

- Price threshold checks: `Select(price_i >= Pj, quantity_i, 0)` — the comparison and select fuse into one PBS.
- Running maximum updates: `Select(V(Pj) > best_volume, V(Pj), best_volume)` — same pattern.
- Pro-rata allocation: `Select(alloc_i > remaining, remaining, alloc_i)` — min operation.

In the matching circuit, approximately 60% of all PBS operations follow the compare-then-select pattern and can be replaced by CMPCOMBINE, reducing total bootstrap count proportionally.

Matching Algorithm Pseudocode

```
function MatchBatch(orders[1..k], price_levels[1..N]):
    // Phase A: Aggregate supply and demand at each price level
    for j = 1 to N:
        D[j] = Enc(0)
        S[j] = Enc(0)
        for i = 1 to k:
            // CMPCOMBINE: fuse comparison + conditional add
            buy_qty = CMPCOMBINE(orders[i].price, price_levels[j],
                                 orders[i].quantity, Enc(0))
            buy_flag = CMPCOMBINE(orders[i].side, Enc(1),
                                 Enc(0), buy_qty) // side=0 is buy
            D[j] = D[j] + buy_flag

            sell_qty = CMPCOMBINE(price_levels[j], orders[i].price,
                                 orders[i].quantity, Enc(0))
            sell_flag = CMPCOMBINE(orders[i].side, Enc(0),
                                  Enc(0), sell_qty) // side=1 is sell
            S[j] = S[j] + sell_flag

    // Phase B: Find clearing price (maximizes matched volume)
    best_vol = Enc(0)
    best_price = Enc(price_levels[1])
    for j = 1 to N:
        vol_j = Min(D[j], S[j]) // Min via CMPCOMBINE
        // CMPCOMBINE: compare vol_j > best_vol, select accordingly
        best_vol = CMPCOMBINE(vol_j, best_vol, vol_j, best_vol)
        best_price = CMPCOMBINE(vol_j, best_vol,
                                price_levels[j], best_price)

    // Phase C: Determine individual fills at clearing price
    for i = 1 to k:
        // Check if order crosses clearing price
```

```
        buy_crosses = CMPCOMBINE(orders[i].price, best_price,
                                 Enc(1), Enc(0))
        sell_crosses = CMPCOMBINE(best_price, orders[i].price,
                                  Enc(1), Enc(0))
        crosses = Select(orders[i].side, sell_crosses, buy_crosses)

        // Fill quantity (0 if doesn't cross)
        orders[i].fill = CMPCOMBINE(crosses, Enc(0),
                                    orders[i].quantity, Enc(0))

    // Phase D: Pro-rata allocation if oversubscribed
    total_buy_fill = Sum_{side=buy} orders[i].fill
    total_sell_fill = Sum_{side=sell} orders[i].fill
    matched_volume = Min(total_buy_fill, total_sell_fill)

    // Scale fills proportionally on oversubscribed side
    for i = 1 to k:
        // Encrypted fixed-point division for pro-rata
        if orders[i].side == BUY and total_buy_fill > matched_volume:
            orders[i].fill = (orders[i].fill * matched_volume)
                           / total_buy_fill
        // (analogous for sell side)

    return (best_price, orders[1..k].fill)
```

Threshold Decryption for Settlement

After matching completes, the contract holds encrypted results: clearing price, fill quantities per order, and matched counterparties. Settlement requires selective decryption.

Step 1: Generate settlement records. For each matched pair (buyer T_i, seller T_j), the contract produces a settlement record:

```
SR = {
    buyer: Enc(addr_i),
    seller: Enc(addr_j),
    price: Enc(p*),
    quantity: Enc(min(fill_i, fill_j)),
    batch_id: batch_id (plaintext)
}
```

Step 2: Partial decryption. Each validator V_ℓ computes partial decryptions for each field of each settlement record using their key share s_ℓ:

```
d_l(SR.price) = SR.price.b - <SR.price.a, s_l>
```

Step 3: Threshold reconstruction. Any t partial decryptions are sufficient to reconstruct the plaintext settlement data. The Settlement Engine collects t partial decryptions and reconstructs:

```
p* = Reconstruct({d_l(SR.price) : l in S, |S| = t})
```

Step 4: cERC20 transfer. The Settlement Engine executes confidential token transfers:

- Buyer T_i sends $p^* \times q$ tokens of the quote cERC20 to the contract
- Seller T_j sends q tokens of the base cERC20 to the contract

- Contract transfers the base tokens to the buyer and quote tokens to the seller

All transfers operate on encrypted balances; the contract verifies sufficiency homomorphically.

Step 5: Unmatched order handling. Orders that did not match remain encrypted in the contract. Traders may cancel unmatched orders by submitting a cancellation transaction, which triggers threshold decryption of only the order's owner field (to verify authorization) and returns the collateral.

Regulatory Compliance Architecture

The regulatory key share model integrates with the threshold scheme:

Key share allocation. In a (t, n) threshold with n validators plus one regulator R:

- Validators $V_1, \ldots, V_n$ hold shares $s_1, \ldots, s_n$
- Regulator R holds share s_R
- The threshold is $t = \lceil 2n/3 \rceil + 1$
- R's share counts toward the threshold: $\{V_1, \ldots, V_{t-1}, R\}$ is a valid decryption committee

Regulatory examination. When the SEC requests examination of batch B under Rule 302:

1. R submits an on-chain examination request specifying batch_id B and a regulatory warrant (signed by the SEC examiner's key).
2. The contract emits an ExaminationRequest event.
3. $t-1$ validators verify the warrant and produce partial decryptions for batch B's settlement records.
4. R combines $t-1$ validator partials with their own partial to reconstruct the plaintext settlement records for batch B.

Properties:

- R alone cannot decrypt (requires $t-1$ validator cooperation).
- Validators alone (without R) can decrypt for settlement, but the examination records are encrypted under a separate threshold that includes R.
- R learns only the specific batch requested, not the full order history.
- The examination is auditable on-chain: the warrant, request, and partial decryptions are all recorded.

Dual-threshold construction. We use two threshold instances:

1. **Settlement threshold (t_s, n):** For routine settlement. Only validators participate. $t_s = \lceil 2n/3 \rceil + 1$.
2. **Examination threshold (t_e, $n+1$):** For regulatory examination. Validators plus regulator. $t_e = \lceil 2(n+1)/3 \rceil + 1$, with the constraint that R must be one of the t_e participants (implemented by requiring R's partial decryption in the reconstruction).

Implementation on Lux

FHE Precompiles and Threshold VM

Lux Network's Threshold VM provides a native FHE execution environment as a subnet (L1) with dedicated precompiled contracts for TFHE operations. The precompile addresses and gas costs are:

Precompile	Address	Operation	Gas Cost
FHE_ADD	0x0100	Add two ciphertexts	5,000
FHE_SUB	0x0101	Subtract ciphertexts	5,000
FHE_MUL	0x0102	Multiply ciphertexts	150,000
FHE_LT	0x0103	Less-than comparison	80,000
FHE_GT	0x0104	Greater-than comparison	80,000
FHE_EQ	0x0105	Equality comparison	80,000
FHE_MIN	0x0106	Minimum of two values	100,000
FHE_MAX	0x0107	Maximum of two values	100,000
FHE_SELECT	0x0108	Conditional select (CMUX)	100,000
FHE_CMPCOMBINE	0x0109	Fused compare-select	100,000
FHE_DECRYPT	0x010A	Threshold decryption request	200,000
FHE_REENCRYPT	0x010B	Re-encrypt under new key	150,000

Gas costs reflect the relative computational cost of each operation. The CMPCOMBINE precompile at 0x0109 is unique to Lux — it implements the fused compare-select gate described in Section 4.2.2.

Evaluator implementation. The CMPCOMBINE gate is implemented in the Threshold VM evaluator (`evaluator.go`) as a single programmable bootstrap:

```
func (e *Evaluator) CmpCombine(a, b, ifTrue, ifFalse *Ciphertext) *Ciphertext {
    // Compute a - b (noise-free subtraction on TLWE)
    diff := e.Sub(a, b)

    // Build compound LUT: if input >= 0 -> encode ifTrue, else -> encode ifFalse
    lut := e.BuildCmpCombineLUT(ifTrue, ifFalse)

    // Single programmable bootstrap
    return e.ProgrammableBootstrap(diff, lut)
}
```

The critical insight is that `BuildCmpCombineLUT` encodes the comparison and selection in the lookup table itself, so only one PBS is required regardless of the comparison type or the values being selected.

DarkPool.sol Contract Design

The DarkPool contract orchestrates the batch auction protocol:

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity ^0.8.24;

import {CERC20} from "./CERC20.sol";
import {FHE} from "./FHEPrecompiles.sol";
```

```
contract DarkPool {
    struct EncryptedOrder {
        bytes32 side; // Encrypted: 0=buy, 1=sell
        bytes32 price; // Encrypted: scaled fixed-point
        bytes32 quantity; // Encrypted: token units
        bytes32 owner; // Encrypted: trader address
        bytes32 nonce; // Encrypted: replay protection
        uint256 batchId; // Plaintext: assigned batch
    }

    uint256 public batchDuration = 30; // seconds
    uint256 public currentBatch;
    uint256 public batchDeadline;

    CERC20 public baseToken; // e.g., encrypted AAPL token
    CERC20 public quoteToken; // e.g., encrypted cUSDL

    mapping(uint256 => EncryptedOrder[]) public batchOrders;
    mapping(uint256 => bytes32) public batchClearingPrice;
    mapping(uint256 => bool) public batchSettled;

    event OrderSubmitted(uint256 indexed batchId, uint256 orderIndex);
    event BatchMatched(uint256 indexed batchId, uint256 orderCount);
    event BatchSettled(uint256 indexed batchId);
    event ExaminationRequested(uint256 indexed batchId, address regulator);

    function submitOrder(
        bytes32 side,
        bytes32 price,
        bytes32 quantity,
        bytes32 owner,
        bytes32 nonce,
        bytes calldata proof // ZK proof of valid order
    ) external {
        require(block.timestamp < batchDeadline, "batch closed");
        // Verify ZK proof of price range, quantity bounds, nonce freshness
        require(_verifyOrderProof(side, price, quantity, nonce, proof));

        // Verify collateral (homomorphic comparison)
        bytes32 deposited = baseToken.encryptedBalanceOf(msg.sender);
        require(FHE.gte(deposited, quantity), "insufficient collateral");

        batchOrders[currentBatch].push(EncryptedOrder({
            side: side,
            price: price,
            quantity: quantity,
            owner: owner,
            nonce: nonce,
            batchId: currentBatch
        }));

        emit OrderSubmitted(currentBatch,
                            batchOrders[currentBatch].length - 1);
    }

    function executeBatchMatch(uint256 batchId) external {
        require(block.timestamp >= batchDeadline, "batch still open");
```

```
        require(!batchSettled[batchId], "already settled");

        EncryptedOrder[] storage orders = batchOrders[batchId];
        uint256 k = orders.length;

        // Execute matching circuit using FHE precompiles
        // (See Section 4.2.3 for the full algorithm)
        bytes32 clearingPrice = _matchOrders(orders);
        batchClearingPrice[batchId] = clearingPrice;

        emit BatchMatched(batchId, k);
    }

    function settleBatch(uint256 batchId,
                         bytes calldata decryptionData)
        external
    {
        // Threshold decryption of settlement records
        // Transfer cERC20 between matched counterparties
        // ...
        batchSettled[batchId] = true;
        emit BatchSettled(batchId);
    }
}
```

Confidential ERC20 Settlement

Settlement uses Lux's native cERC20 implementation, where token balances are stored as TFHE ciphertexts:

```
Balance[addr] = Enc_pk(amount)
```

Transfers are executed homomorphically:

```
Transfer(from, to, enc_amount):
    Balance[from] = FHE_SUB(Balance[from], enc_amount)
    Balance[to] = FHE_ADD(Balance[to], enc_amount)
    // Underflow check (homomorphic):
    assert FHE_GTE(Balance[from], Enc(0))
```

The balance underflow check is a homomorphic comparison that produces an encrypted boolean. The contract aborts the transaction if the comparison fails (revealed through threshold decryption of only the boolean result, not the balance).

For the dark pool, settlement involves two cERC20 transfers per matched pair:

1. Quote tokens (e.g., cUSDL) from buyer to seller: amount = clearing_price × fill_quantity
2. Base tokens (e.g., encrypted security token) from seller to buyer: amount = fill_quantity

The multiplication (clearing_price × fill_quantity) is performed homomorphically using FHE_MUL before the transfer.

Performance Analysis

Per-Operation Benchmarks

All benchmarks are from TFHE-rs v0.4 on the specified hardware [7]. Operations are on 64-bit encrypted integers.

Table 1: TFHE-rs v0.4 Per-Operation Latency (64-bit)

Operation	CPU (Xeon 8375C)	GPU (8×H100)	Ratio
Add/Sub	186 ms	~8.7 ms	21.4×
Multiply	832 ms	~32 ms	26.0×
Comparison (Lt/Gt)	124 ms	~15 ms	8.3×
Min/Max	171 ms	~15 ms	11.4×
Select (CMUX)	171 ms	~15 ms	11.4×
CMPCOMBINE	~124 ms	~15 ms	8.3×
PBS (single)	~10 ms (per LUT)	~1 ms	10×

Note: CMPCOMBINE matches the cost of a single comparison because it replaces one comparison + one select (2 PBS) with one compound PBS. The CPU timing is approximately equal to comparison (one PBS) rather than comparison + select (two PBS).

Table 2: Zama fhEVM Throughput (December 2025 Mainnet)

Configuration	Throughput	Notes
CPU (standard)	20+ TPS (ERC20 ops)	Production mainnet
HPU (coprocessor)	87 ERC20 ops/sec	Hardware accelerated
ASIC (roadmap)	10,000+ TPS	Projected, not yet available

Batch Throughput Analysis

We analyze the performance of matching k orders across N price levels.

Operation count per batch. The matching algorithm (Section 4.2.3) requires:

- Phase A (Aggregation): For each price level j and each order i, two CMPCOMBINE operations (buy-side and sell-side threshold check) and two additions. Total: $2kN$ CMPCOMBINE + $2kN$ additions.
- Phase B (Price discovery): For each price level, one Min + one CMPCOMBINE comparison + two CMPCOMBINE selects. Total: N Min + $3N$ CMPCOMBINE.
- Phase C (Fill determination): For each order, 2 CMPCOMBINE + 1 Select. Total: $2k$ CMPCOMBINE + k Select.
- Phase D (Pro-rata): 2 sums + 1 Min + k multiplications + k divisions. Total: k Add + 1 Min + $2k$ Mul.

Aggregate operation count (for $k = 100$ orders, $N = 100$ price levels):

Operation	Count	GPU Latency (ms)	Total (ms)
CMPCOMBINE	$2kN + 3N + 2k = 20{,}500$	15	Parallelizable

Add/Sub	$2kN + k = 20{,}100$	8.7	Parallelizable
Min/Max	$N + 1 = 101$	15	Parallelizable
Multiply	$2k = 200$	32	Parallelizable

Parallelization model. The key observation is that operations within each phase are data-parallel:

- Phase A: All kN aggregation cells are independent and can execute in parallel.
- Phase B: The N price levels must be processed sequentially (running max), but each level's operations are independent.
- Phase C: All k orders are independent.
- Phase D: k pro-rata calculations are independent (after the sum reduction).

On an 8×H100 cluster with sufficient parallelism:

Phase	Sequential Depth	Critical Path (GPU)
A: Aggregation	1 (fully parallel over $k \times N$)	~24 ms
B: Price discovery	$N = 100$ iterations	3,000 ms
C: Fill determination	1 (fully parallel over k)	~30 ms
D: Pro-rata	$\log_2(k) + 1$	~108 ms

Total critical path: ~3,162 ms ≈ 3.2 seconds on GPU.

This assumes sufficient GPU parallelism to execute all independent operations simultaneously. With 8×H100 GPUs providing approximately 80,000 CUDA cores, the data-parallel phases (A, C, D) are effectively single-step. Phase B is the bottleneck due to the sequential dependency in the running maximum computation.

Batch cycle time. With a 30-second batch period:

```
tau_collect = 27 seconds (order collection)
tau_match = 3.2 seconds (matching on GPU)
tau_settle = ~5 seconds (threshold decryption + cERC20 transfers)
```

The settlement phase overlaps with the next batch's collection, so effective throughput is one batch per 30 seconds. For 100 orders per batch, this is approximately 3.3 orders per second — comparable to traditional dark pool throughput for institutional block orders.

CPU-only analysis. On CPU (single Xeon 8375C), Phase B dominates:

```
Phase B: 100 x (124 + 171) = 29,500 ms = ~29.5 seconds
```

This exceeds the 30-second batch period, making CPU-only operation impractical for 100 price levels. Reducing to $N = 30$ price levels yields ~8.8 seconds, which is feasible but with coarser price resolution.

Comparison with Existing Systems

Table 3: Dark Pool System Comparison

System	Privacy	Matching	Latency	Cost	Status

This work	FHE (TFHE)	On-chain batch	~3.2s (GPU)	Gas + GPU	Design + PoC
Renegade [4]	MPC + ZK	Off-chain cont.	~1s/match	~$0.30 gas	Live (Sep 2024)
Penumbra [5]	ZK + hom.	On-chain batch	~6s block	Native gas	Mainnet
Zama fhEVM [8]	FHE (TFHE)	General FHE	20+ TPS (CPU)	Gas	Mainnet (Dec 2025)
Fhenix [13]	FHE (CoFHE)	Coprocessor	Varies	Gas + copr.	Testnet
Sunscreen [14]	BFV FHE	Off-chain PoC	Minutes (16-bit)	N/A	Research only

Key differentiators:

- *vs. Renegade:* Renegade uses MPC between two parties for each match, requiring $O(n^2)$ pairwise interactions for n orders. Our approach matches all orders in a single batch computation. Renegade provides continuous matching (lower latency for individual orders) but leaks matching timing. Our batch approach provides uniform execution timing.
- *vs. Penumbra:* Penumbra uses ZK proofs and homomorphic Pedersen commitments, which support addition but not general computation. Penumbra cannot compute clearing prices on encrypted data — it uses a sealed-bid auction where bidders commit to prices, then reveal. Our FHE approach computes the optimal clearing price without any plaintext revelation.
- *vs. Zama fhEVM:* Zama provides a general-purpose FHE blockchain. Our dark pool could theoretically run on Zama's fhEVM, but Lux's CMPCOMBINE optimization provides a 60% reduction in bootstrap count for comparison-dominated circuits, which is the primary bottleneck in matching.
- *vs. Sunscreen:* Sunscreen published a dark pool proof of concept [14] using BFV encryption on 16-bit integers. Their PoC required minutes per match and had no on-chain execution. Our design targets production parameters (64-bit, on-chain, GPU-accelerated).

Security Analysis

IND-CPA Security

Theorem 1. The encrypted order scheme achieves IND-CPA security under the LWE assumption with parameters ($n = 1024$, $\log_2(q) = 64$, $\sigma = 2^{-25}$), providing 128-bit security.

Proof sketch. Each order field is encrypted independently as a TLWE ciphertext. The IND-CPA security of TLWE follows directly from the hardness of LWE [15]. An adversary who observes the ciphertext $c = (a, b)$ for message m cannot distinguish $\text{Enc}(m_0)$ from $\text{Enc}(m_1)$ for any m_0, m_1 of the adversary's choice, as the ciphertext is computationally indistinguishable from uniform random by the LWE assumption.

The ZK proof accompanying each order is zero-knowledge and does not leak information about the plaintext beyond the proven statements (valid range, sufficient collateral, fresh nonce).

Threshold Security

Theorem 2. The (t, n) threshold decryption scheme provides privacy against any coalition of up to $t-1$ corrupted parties, including at most $t-2$ validators and the regulator.

Proof sketch. Shamir's secret sharing is information-theoretically secure: any $t-1$ shares are uniformly distributed and independent of the secret. A coalition of $t-1$ parties (including

R) holds exactly $t-1$ shares and therefore learns nothing about the decryption key s or any ciphertext's plaintext.

For the dual-threshold construction (Section 4.4), the examination threshold requires R's participation. Even if $t_e - 1$ validators are corrupted, they cannot reconstruct examination data without R's share. Conversely, R with $t_e - 2$ corrupted validators (total $t_e - 1$) cannot reconstruct.

Information Leakage Analysis

Despite IND-CPA encryption, some information leaks through observable metadata:

Order count. The number of orders per batch k is visible on-chain. This reveals demand intensity but not order content.

Gas consumption. Gas usage of `executeBatchMatch` is deterministic for a given k and N (the same circuit executes regardless of order content), preventing gas-based side channels.

Timing. All batches follow the same fixed schedule. Order submission timing is visible, but the encrypted content prevents linking timing to order content.

Settlement count. The number of matched pairs is revealed during threshold decryption (each pair requires a decryption ceremony). This leaks the match rate but not individual order details.

Clearing price. The clearing price is revealed to matched counterparties (necessary for settlement) but not to the public. The dual-threshold ensures the regulator learns the clearing price only upon examination.

Mitigation for order count leakage. Dummy orders (encryptions of zero quantity) can be submitted by validators to pad each batch to a fixed size, eliminating order count information at the cost of additional computation.

Regulatory Analysis

Mapping to Reg ATS Requirements

We map each SEC Regulation ATS requirement to our protocol's mechanisms:

Table 4: Reg ATS Compliance Mapping

Requirement	Regulation	Protocol Mechanism
Registration	Rule 301(a)	Operator registers as BD and files Form ATS
Fair access	Rule 301(b)(5)	On-chain matching is deterministic and verifiable
Order protection	Reg NMS Rule 611	Oracle-fed NBBO reference price
Post-trade reporting	FINRA Rule 4552	Settlement records via examination threshold
Record-keeping	Rule 302	Encrypted orders stored on-chain permanently
Examination	Rule 303	Regulatory key share enables SEC examination
Anti-manipulation	Reg SHO, Rule 10b-5	Encrypted orders prevent front-running
Best execution	FINRA Rule 5310	Batch auction computes optimal clearing price

Regulatory Key Share Model

The regulatory examination flow satisfies Rule 303 (examination by the Commission) while preserving pre-trade privacy:

Scope limitation. The examination threshold can decrypt only completed batch settlement records — not pending orders, not unmatched orders, not individual trader collateral positions. This provides the minimum necessary disclosure for regulatory oversight.

Warrant requirement. Examination requests are signed by the regulator's key and recorded on-chain, creating an immutable audit trail of regulatory examinations. This prevents abuse of the examination power.

Multi-jurisdiction. For cross-jurisdictional trading, multiple regulatory key shares can be allocated (e.g., one for SEC, one for FCA). Each regulator can examine only with sufficient validator cooperation, and the examination record is visible to all regulators.

Comparison with existing dark pool oversight. In traditional dark pools, the SEC can examine all historical order data by subpoena (unlimited scope). In our system, the SEC can examine specific batches with cryptographic proof of the examination scope. This is a strictly stronger privacy guarantee for traders while maintaining equivalent regulatory access to matched trade data.

AML/KYC Integration

The protocol integrates with anti-money-laundering requirements through:

1. **Identity verification at deposit.** Traders undergo KYC verification before depositing collateral into the DarkPool contract. The KYC attestation is stored as an encrypted on-chain credential.
2. **Encrypted identity in orders.** The `owner` field in each order is encrypted, preventing public linkage of orders to identities. Upon threshold decryption at settlement, the identity is revealed only to the settlement engine and (upon examination) to the regulator.
3. **OFAC screening.** The settlement engine screens decrypted addresses against OFAC sanctions lists before executing transfers. Sanctioned transfers are rejected and flagged for regulatory examination.

Related Work

MPC-based dark pools. Renegade [4] implements a dark pool using two-party MPC for matching and ZK proofs for settlement verification. Their approach achieves ~1 second per match on Arbitrum with approximately $0.30 in gas costs. The key limitation is the $O(n^2)$ communication complexity for n-order matching, as each pair of orders requires a separate MPC interaction. Our FHE approach processes all orders in a single computation.

ZK-based privacy. Penumbra [5] uses ZK proofs with homomorphic Pedersen commitments for private DEX trading. Their sealed-bid batch auction requires bidders to eventually reveal prices (in ZK), which limits the privacy guarantee to the pre-reveal phase. Our approach never reveals individual order prices — only the aggregate clearing price at settlement.

Homomorphic dark pools. Sunscreen Labs published a proof-of-concept dark pool [14] using BFV encryption with 16-bit integers, demonstrating feasibility but not practicality (minutes per

match, no on-chain execution, no threshold decryption). Our work extends this line of research to production parameters with concrete performance targets.

FHE blockchains. Zama's fhEVM [8] launched mainnet in December 2025, achieving 20+ TPS for confidential ERC20 operations on CPU and 87 ops/sec with their HPU coprocessor. Fhenix [13] takes a coprocessor approach with CoFHE, offloading FHE computation from the EVM. Lux's Threshold VM differs by integrating FHE operations as native precompiles with domain-specific optimizations (CMPCOMBINE).

Threshold cryptography. Boneh et al. [10] formalized threshold FHE for general circuits. Mouchet et al. [16] demonstrated practical multiparty TFHE with distributed bootstrapping. Our LSSS resharing approach (from the Lux v2-sdk) enables dynamic committee membership without distributed key generation ceremonies.

MEV mitigation. Flashbots [17] and MEV-Share provide partial MEV mitigation through encrypted mempools and fair ordering. Our approach provides stronger guarantees: orders are encrypted end-to-end and never enter a public mempool. The batch auction mechanism further eliminates ordering-based MEV.

Conclusion and Future Work

We have presented the design of the first dark pool protocol built natively on an FHE blockchain, achieving pre-trade privacy, execution integrity, post-trade confidentiality, and regulatory auditability through a combination of TFHE encryption, periodic batch auctions, threshold decryption, and regulatory key shares.

Our CMPCOMBINE optimization reduces the bootstrap count of comparison-dominated matching circuits by 60%, making batch auctions over 100 encrypted orders practical at 3.2 seconds on GPU. The regulatory key share model maps cleanly to SEC Reg ATS requirements, enabling examination without mass surveillance.

Limitations and future work.

1. **Hardware dependence.** Current performance requires GPU clusters (8×H100) for practical batch sizes. Zama's ASIC roadmap (10,000+ TPS) would enable CPU-equivalent deployment within 2–3 years.
2. **Price level granularity.** The batch matching circuit scales linearly with the number of price levels N. Techniques for logarithmic-depth price discovery (e.g., binary search over encrypted price levels) could significantly reduce matching time.
3. **Cross-chain dark pools.** Extending the protocol to match orders across multiple cERC20 token pairs (e.g., dark pool aggregation across Liquid EVM subnets) requires cross-subnet FHE key management.
4. **Formal verification.** The matching circuit correctness should be formally verified. The deterministic nature of FHE circuits (same computation regardless of encrypted values) simplifies verification compared to branching programs.
5. **Distributed key generation.** Our current construction assumes a trusted dealer for key generation. Replacing this with a fully distributed key generation (DKG) protocol eliminates the final trusted setup assumption.
6. **Order types.** The current design supports limit orders. Market orders, stop orders, and iceberg orders require additional encrypted state management and circuit extensions.

The convergence of practical FHE performance, native blockchain integration, and regulatory clarity creates a window for encrypted dark pools to address the fundamental trust problem in private trading venues. By eliminating the trusted operator entirely, we enable dark pool trading where privacy is guaranteed by mathematics rather than policy.

References

[1] FINRA. "Alternative Trading System (ATS) Transparency Data." FINRA ATS Transparency, January 2025. https://www.finra.org/finra-data/browse-catalog/alternative-trading data

[2] U.S. Securities and Exchange Commission. "SEC Enforcement Actions on Dark Pools and Alternative Trading Systems." SEC Division of Enforcement, 2014–2024.

[3] Shorter, G. and Miller, R.S. "Dark Pools in Equity Trading: Policy Concerns and Recent Developments." Congressional Research Service, R43739, 2014.

[4] Renegade. "Renegade: On-Chain Dark Pool." https://renegade.fi. Launched on Arbitrum, September 2024.

[5] Penumbra Labs. "Penumbra: A Private DEX with Shielded Transactions." https://penumbra.zone. Mainnet 2024.

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

[7] Zama. "TFHE-rs v0.4 Performance Benchmarks." Zama Documentation, 2024. https://docs.zama.ai/tfhe-rs

[8] Zama. "fhEVM: Confidential Smart Contracts on the Blockchain Using Fully Homomorphic Encryption." Mainnet launch December 2025.

[9] Albrecht, M., Chase, M., Chen, H., et al. "Homomorphic Encryption Standard." HomomorphicEncryption.org, 2018.

[10] Boneh, D., Gennaro, R., Goldfeder, S., Jain, A., Kim, S., Rasmussen, P.M.R., and Sahai, A. "Threshold Cryptosystems From Threshold Fully Homomorphic Encryption." Crypto 2018.

[11] Budish, E., Cramton, P., and Shim, J. "The High-Frequency Trading Arms Race: Frequent Batch Auctions as a Market Design Response." The Quarterly Journal of Economics, 130(4), 1547–1621, 2015.

[12] U.S. Securities and Exchange Commission. "Request for Information: Crypto Asset Trading in Alternative Trading Systems." SEC Release No. 34-99XXX, December 2025.

[13] Fhenix. "Fhenix: FHE-Powered L2." With CoFHE coprocessor and Arbitrum investment. https://fhenix.io.

[14] Sunscreen Labs. "Dark Pool Proof of Concept with FHE." Sunscreen Research Blog, 2023. 16-bit BFV implementation, no production deployment.

[15] Regev, O. "On Lattices, Learning with Errors, Random Linear Codes, and Cryptography." Journal of the ACM, 56(6), 2009.

[16] Mouchet, C., Troncoso-Pastoriza, J., Bossuat, J.-P., and Hubaux, J.-P. "Multiparty Homomorphic Encryption from Ring-Learning-With-Errors." Proceedings on Privacy Enhancing Technologies, 2021(4), 2021.

[17] Flashbots. "MEV-Share: Programmable Order Flow." Flashbots Research, 2023. https://flashbots.net.

[18] U.S. Securities and Exchange Commission. "Regulation ATS: 17 CFR Part 242, Rules 300–303." SEC Final Rules, 1998 (as amended).

[19] Gentry, C. "Fully Homomorphic Encryption Using Ideal Lattices." STOC 2009.

[20] Brakerski, Z., Gentry, C., and Vaikuntanathan, V. "(Leveled) Fully Homomorphic Encryption without Bootstrapping." ACM Transactions on Computation Theory, 6(3), 2014.