Lattice Cryptography from First Principles: A Builder's Guide

Zach Kelling

Lux Industries

2025

Abstract

Lattice-based cryptography is the foundation of post-quantum security: ML-DSA for signatures, ML-KEM for key encapsulation, and Pulsar for threshold schemes all build on the hardness of lattice problems. This paper teaches lattice cryptography from the ground up—not as a survey of results, but as a builder's guide for engineers implementing these schemes. We cover: lattices and their geometry, the Learning With Errors (LWE) problem and why it is hard, Ring-LWE and Module-LWE for practical efficiency, the Number Theoretic Transform (NTT) for polynomial arithmetic, how ML-DSA and ML-KEM work step by step, and how Pulsar extends these to threshold operations. We include Go code for every non-trivial algorithm and explain every design decision in terms of the security/performance tradeoff it represents. This is the paper we wish existed when we started implementing post-quantum cryptography for Lux Network.

Contents

1 What Is a Lattice

Definition 1 (Lattice)**.** *A lattice* Λ *in* $\mathbb{R}^n$ *is the set of all integer linear combinations of a set of linearly independent vectors* $\mathbf{b}_1, \ldots, \mathbf{b}_n \in \mathbb{R}^n$*:*

$$\Lambda = \left\{ \sum_{i=1}^{n} z_i \mathbf{b}_i \ : \ z_i \in \mathbb{Z} \right\}$$

The vectors $\{\mathbf{b}_i\}$ *form a* basis *for* Λ*.*

Think of a lattice as a grid—a regular arrangement of points in space. In two dimensions, it looks like graph paper tilted and stretched. The crucial property: the grid is *discrete* (points are separated by gaps) but *infinite* (it extends in all directions).

1.1 Why Lattices for Cryptography

Classical public-key cryptography (RSA, elliptic curves) relies on the hardness of factoring or discrete logarithms. Shor's algorithm solves both in polynomial time on a quantum computer. Lattice problems—finding short vectors in high-dimensional lattices—resist all known quantum algorithms.

The two fundamental hard problems:

Definition 2 (Shortest Vector Problem (SVP))**.** *Given a lattice basis* $\mathbf{B}$*, find the shortest nonzero lattice vector.*

Definition 3 (Closest Vector Problem (CVP))**.** *Given a lattice basis* $\mathbf{B}$ *and a target point* $\mathbf{t}$*, find the lattice vector closest to* $\mathbf{t}$*.*

Both are NP-hard in their exact forms, and the best known algorithms (classical or quantum) for approximate versions run in exponential time in the lattice dimension n.

1.2 The Geometry of Hard Problems

In low dimensions ($n = 2, 3$), finding short vectors is easy—look at the basis and reduce it. In high dimensions ($n = 256, 512, 1024$), the geometry becomes pathological: the shortest vector is exponentially shorter than the basis vectors, but finding it requires exploring an exponentially large search space. This is the core intuition: lattice cryptography works because high-dimensional geometry is hard.

2 Learning With Errors (LWE)

2.1 The Problem

Definition 4 (Learning With Errors)**.** *Given* m *samples of the form* $(\mathbf{a}_i, b_i)$ *where:*

- $\mathbf{a}_i \in \mathbb{Z}_q^n$ *is uniformly random,*
- $b_i = \langle \mathbf{a}_i, \mathbf{s} \rangle + e_i \pmod{q}$*,*
- $\mathbf{s} \in \mathbb{Z}_q^n$ *is a secret vector,*
- e_i *is a small "error" sampled from a discrete Gaussian,*

recover $\mathbf{s}$*.*

Without the error e_i, this is a system of linear equations—solvable by Gaussian elimination. The error makes the problem hard: it turns an easy linear algebra problem into a problem equivalent to finding short vectors in a lattice.

2.2 Why LWE Is Hard

Regev (2005) proved that solving LWE is at least as hard as solving worst-case lattice problems (approximate SVP and approximate SIVP) in dimension n. This is a *worst-case to average-case* reduction: even random LWE instances are as hard as the hardest lattice instances. No other family of cryptographic assumptions has this property.

2.3 LWE as Encryption

LWE directly yields a public-key encryption scheme:

- **Key generation**: Choose secret $\mathbf{s}$. Publish samples $(\mathbf{A}, \mathbf{b} = \mathbf{A}\mathbf{s} + \mathbf{e})$.
- **Encryption of bit** m: Choose random $\mathbf{r}$. Compute $\mathbf{u} = \mathbf{A}^T\mathbf{r}$ and $v = \mathbf{b}^T\mathbf{r} + m \cdot \lfloor q/2 \rfloor$.
- **Decryption**: Compute $v - \mathbf{u}^T\mathbf{s} = m \cdot \lfloor q/2 \rfloor + \text{small noise}$. Round to recover m.

The decryption works because the noise terms are small relative to $q/2$. The scheme is secure because distinguishing $(\mathbf{A}, \mathbf{A}\mathbf{s} + \mathbf{e})$ from uniform random is the LWE problem.

3 Ring-LWE and Module-LWE

3.1 The Efficiency Problem

Plain LWE has a fundamental efficiency issue: the public key is a matrix $\mathbf{A} \in \mathbb{Z}_q^{m \times n}$, which has size $O(n^2)$. For 128-bit security, $n \approx 512$, so keys are hundreds of kilobytes. This is impractical for blockchain use.

3.2 Ring-LWE

Ring-LWE replaces vectors over $\mathbb{Z}_q$ with elements of a polynomial ring:

$$R_q = \mathbb{Z}_q[X]/(X^n + 1)$$

where n is a power of 2 and q is prime. An element of R_q is a polynomial of degree $< n$ with coefficients in $\mathbb{Z}_q$.

Definition 5 (Ring-LWE)**.** *Given samples* $(a_i, b_i = a_i \cdot s + e_i) \in R_q \times R_q$*, recover* $s \in R_q$*.*

The key advantage: a single ring element replaces an entire vector, reducing key sizes from $O(n^2)$ to $O(n)$. Polynomial multiplication in R_q replaces matrix-vector multiplication, and NTT makes it $O(n \log n)$ instead of $O(n^2)$.

3.3 Module-LWE

Module-LWE (M-LWE) is a middle ground: it uses *small matrices of ring elements* rather than single ring elements or large integer matrices.

Definition 6 (Module-LWE)**.** *Let* k *be a small integer (2, 3, or 4). Given* $(\mathbf{A}, \mathbf{b} = \mathbf{A}\mathbf{s} + \mathbf{e})$ *where* $\mathbf{A} \in R_q^{k \times k}$*,* $\mathbf{s}, \mathbf{e} \in R_q^k$*, recover* $\mathbf{s}$*.*

M-LWE is the basis for ML-KEM and ML-DSA. The module dimension k parameterizes the security level:

Scheme	n	k	q	Security
ML-KEM-512 (Kyber-512)	256	2	3329	128-bit
ML-KEM-768 (Kyber-768)	256	3	3329	192-bit
ML-KEM-1024 (Kyber-1024)	256	4	3329	256-bit
ML-DSA-44 (Dilithium2)	256	4/4	8380417	128-bit
ML-DSA-65 (Dilithium3)	256	6/5	8380417	192-bit
ML-DSA-87 (Dilithium5)	256	8/7	8380417	256-bit

Table 1: NIST post-quantum standard parameters. All use $n = 256$.

4 The Number Theoretic Transform

Polynomial multiplication in R_q is the bottleneck operation. Naive multiplication is $O(n^2)$. The NTT reduces this to $O(n \log n)$.

4.1 What Is NTT

The NTT is the discrete Fourier transform over a finite field $\mathbb{Z}_q$ instead of $\mathbb{C}$. It transforms a polynomial from coefficient representation to evaluation representation (values at the n-th roots of unity in $\mathbb{Z}_q$).

Definition 7 (NTT)**.** *Given a polynomial $a(X) = \sum_{i=0}^{n-1} a_i X^i$ and a primitive n-th root of unity $\omega \in \mathbb{Z}_q$:*

$$\hat{a}_j = NTT(a)_j = \sum_{i=0}^{n-1} a_i \omega^{ij} \pmod{q}$$

In evaluation representation, polynomial multiplication becomes pointwise:

$$\text{NTT}(a \cdot b)_j = \text{NTT}(a)_j \cdot \text{NTT}(b)_j$$

So: transform both polynomials ($2{\times}O(n \log n)$), multiply pointwise ($O(n)$), inverse-transform the result ($O(n \log n)$). Total: $O(n \log n)$.

4.2 The Butterfly Operation

The NTT is computed using the Cooley-Tukey butterfly, the same structure as the FFT:

Listing 1: NTT butterfly operation in Go

```go
// Butterfly: the core NTT operation
// In-place: a[j], a[j+step] are updated
func butterfly(a []uint32, j, step int, w uint32, q uint32) {
    t := mulmod(w, a[j+step], q)
    a[j+step] = submod(a[j], t, q)
    a[j] = addmod(a[j], t, q)
}

// Full NTT (iterative, in-place, Cooley-Tukey)
func NTT(a []uint32, omegas []uint32, q uint32) {
    n := len(a)
    bitReverse(a) // Reorder elements

    for step := 1; step < n; step <<= 1 {
        for j := 0; j < n; j += step << 1 {
            for k := 0; k < step; k++ {
                butterfly(a, j+k, step, omegas[step+k], q)
```

```
            }
        }
    }
}
```

4.3 Montgomery Multiplication

Modular multiplication ($a \cdot b \mod q$) is expensive because division is slow. Montgomery multiplication avoids division by working in Montgomery form:

Listing 2: Montgomery multiplication

```
const (
    Q      uint32 = 3329        // ML-KEM modulus
    QInv   uint32 = 62209       // Q^{-1} mod 2^16
    Mont   uint32 = 2285        // 2^16 mod Q (Montgomery constant)
)

// Montgomery reduction: given a < Q * 2^16,
// compute a * 2^{-16} mod Q without division
func montgomeryReduce(a uint32) uint16 {
    t := uint16(a) * uint16(QInv)
    return uint16((a - uint32(t)*uint32(Q)) >> 16)
}

// Multiply in Montgomery form
func mulMont(a, b uint16) uint16 {
    return montgomeryReduce(uint32(a) * uint32(b))
}
```

5 How ML-KEM Works

ML-KEM is a key encapsulation mechanism (KEM): it produces a shared secret between two parties. It builds on M-LWE.

5.1 Key Generation

1. Sample random matrix $\mathbf{A} \in R_q^{k \times k}$ (from a seed via SHAKE-128).
2. Sample secret $\mathbf{s} \in R_q^k$ and error $\mathbf{e} \in R_q^k$ with small coefficients (centered binomial distribution).
3. Compute $\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e}$.
4. Public key: (seed for $\mathbf{A}, \mathbf{t}$). Secret key: $\mathbf{s}$.

Listing 3: ML-KEM key generation (simplified)

```
func KeyGen() (PublicKey, SecretKey) {
    seed := randBytes(32)
    A := expandA(seed)  // k x k matrix of ring elements

    s := sampleCBD(eta1) // Secret: small coefficients
    e := sampleCBD(eta1) // Error: small coefficients

    // t = A*s + e (all operations in NTT domain)
    sNTT := nttVector(s)
    t := addVectors(mulMatVec(A, sNTT), nttVector(e))

```

```
    pk := PublicKey{Seed: seed, T: compress(t)}
    sk := SecretKey{S: sNTT}
    return pk, sk
}
```

5.2 Encapsulation

1. Generate random message $m \leftarrow \{0,1\}^{256}$.
2. Derive randomness: $(r, K) = G(m \| H(\text{pk}))$.
3. Encrypt m under the public key using randomness r: $\mathbf{u} = \mathbf{A}^T\mathbf{r} + \mathbf{e}_1$ and $v = \mathbf{t}^T\mathbf{r} + e_2 + \lceil q/2 \rceil \cdot m$.
4. Ciphertext: $(\mathbf{u}, v)$. Shared secret: K.

5.3 Decapsulation

1. Decrypt: $m' = \text{Decompress}(v - \mathbf{s}^T\mathbf{u})$.
2. Re-encrypt: $(\mathbf{u}', v') = \text{Enc}(\text{pk}, m'; r')$.
3. If $(\mathbf{u}', v') = (\mathbf{u}, v)$, output $K = \text{KDF}(m' \| H(\text{ct}))$.
4. Otherwise, output a random key (implicit rejection).

The re-encryption step (Fujisaki-Okamoto transform) converts CPA security to CCA security: it prevents the adversary from learning anything by submitting crafted ciphertexts.

6 How ML-DSA Works

ML-DSA is a digital signature scheme based on the "Fiat-Shamir with aborts" paradigm over Module-LWE.

6.1 Key Generation

1. Sample $\mathbf{A} \in R_q^{k \times \ell}$ from a seed.
2. Sample secret vectors $\mathbf{s}_1 \in R_q^{\ell}$, $\mathbf{s}_2 \in R_q^{k}$ with small coefficients.
3. Compute $\mathbf{t} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$.
4. Public key: $(\text{seed}, \mathbf{t})$. Secret key: $(\mathbf{s}_1, \mathbf{s}_2)$.

6.2 Signing (Fiat-Shamir with Aborts)

1. Sample random masking vector $\mathbf{y}$ with bounded coefficients.
2. Compute $\mathbf{w} = \mathbf{A}\mathbf{y}$ and extract high bits: $\mathbf{w}_1$.
3. Compute challenge: $c = H(\text{msg} \| \mathbf{w}_1)$.
4. Compute response: $\mathbf{z} = \mathbf{y} + c \cdot \mathbf{s}_1$.
5. **Rejection sampling**: If $\|\mathbf{z}\|_\infty$ is too large, *restart*. This prevents the signature from leaking information about $\mathbf{s}_1$.
6. Output signature: $(c, \mathbf{z})$.

The rejection sampling step is critical: without it, statistical analysis of many signatures reveals $\mathbf{s}_1$. With it, the distribution of $\mathbf{z}$ is independent of $\mathbf{s}_1$ (up to the bound check).

Listing 4: ML-DSA signing with rejection sampling

```
func Sign(sk *SecretKey, msg []byte) Signature {
    for {
        // Sample masking vector
        y := sampleMask(gamma1)

        // w = A * y
```

```
        w := mulMatVec(sk.A, nttVector(y))
        w1 := highBits(w, alpha)

        // Challenge
        c := challengeHash(msg, w1)
        cPoly := sampleChallenge(c)

        // Response
        z := addVectors(y, scalarMulVector(cPoly, sk.S1))

        // Rejection: check norm bound
        if infNorm(z) >= gamma1-beta {
            continue // Restart
        }

        // Check hint
        r0 := lowBits(
            subVectors(w, scalarMulVector(cPoly, sk.S2)),
            alpha,
        )
        if infNorm(r0) >= gamma2-beta {
            continue // Restart
        }

        return Signature{C: c, Z: z}
    }
}
```

6.3 Verification

1. Compute $\mathbf{w}' = \mathbf{Az} - c \cdot \mathbf{t}$.
2. Extract high bits: $\mathbf{w}'_1$.
3. Check: $c = H(\text{msg} \| \mathbf{w}'_1)$ and $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$.

Verification is straightforward and never requires rejection sampling.

7 Pulsar: Threshold Lattice Signatures

Pulsar is Lux Network's threshold signature scheme based on Module-LWE (threshold ML-DSA-65, byte-equal to FIPS-204). It allows t-of-n validators to collaboratively sign messages without any single validator holding the full secret key.

7.1 Design Challenges

Threshold ML-DSA is hard because of rejection sampling: each signer's partial signature must independently pass the norm bound, and the combined signature must also pass. Naive approaches have exponentially low success probability as t grows.

Pulsar addresses this through:

1. **Expanded masking range**: Each signer uses a larger γ_1 to ensure individual rejection rarely triggers.
2. **Aggregation-friendly bounds**: The combined norm bound accounts for the sum of t partial signatures.
3. **Share refresh**: Proactive secret sharing allows periodic refresh of shares without changing the group public key.

7.2 Key Distribution

Listing 5: Pulsar key generation via Shamir's secret sharing over R_q

```
func DistributeKey(n, t int) (PublicKey, []SecretShare) {
    // Generate master secret
    s := sampleSecret()

    // Shamir's secret sharing over the ring R_q
    // f(x) = s + a_1*x + ... + a_{t-1}*x^{t-1}
    coeffs := make([]RingElement, t)
    coeffs[0] = s
    for i := 1; i < t; i++ {
        coeffs[i] = sampleRing()
    }

    // Evaluate polynomial at n points
    shares := make([]SecretShare, n)
    for i := 0; i < n; i++ {
        xi := ringFromScalar(uint32(i + 1))
        shares[i] = SecretShare{
            Index: i + 1,
            Value: evaluatePoly(coeffs, xi),
        }
    }

    // Public key from master secret
    pk := computePublicKey(s)
    return pk, shares
}
```

8 Our Go Implementation

The entire lattice cryptography stack for Lux Network is implemented in Go with SIMD acceleration via assembly (covered in our companion NTT paper).

8.1 Package Structure

Listing 6: Package layout

```
luxfi/crypto/
    lattice/
        ring.go        # Ring element operations
        ntt.go         # NTT forward/inverse
        ntt_amd64.s    # AVX2 NTT butterfly
        poly.go        # Polynomial arithmetic
        sample.go      # Sampling (CBD, uniform, challenge)
    mlkem/
        keygen.go      # ML-KEM key generation
        encaps.go      # Encapsulation
        decaps.go      # Decapsulation
    mldsa/
        keygen.go      # ML-DSA key generation
        sign.go        # Signing with rejection sampling
        verify.go      # Verification
    corona/
```

```
        threshold.go  # Threshold operations
        share.go      # Secret share management
        aggregate.go  # Partial signature aggregation
```

8.2 Performance

Operation	Time	vs. Reference	Platform
ML-KEM-768 KeyGen	28 μs	1.1x ref C	Apple M2
ML-KEM-768 Encaps	34 μs	1.2x ref C	Apple M2
ML-KEM-768 Decaps	38 μs	1.1x ref C	Apple M2
ML-DSA-65 KeyGen	52 μs	1.3x ref C	Apple M2
ML-DSA-65 Sign	186 μs	1.4x ref C	Apple M2
ML-DSA-65 Verify	56 μs	1.2x ref C	Apple M2
NTT-256 (pure Go)	3.1 μs	2.8x ref C	Apple M2
NTT-256 (AVX2)	0.9 μs	0.95x ref C	AMD Zen 4

Table 2: Performance of Lux Network's Go lattice crypto implementation.

9 Security Parameters and Concrete Hardness

9.1 How to Choose Parameters

Lattice parameters are chosen to resist the best known attacks. The primary attack is the *primal lattice attack* using the BKZ algorithm:

1. Construct a lattice from the LWE instance.
2. Run BKZ with block size β to find a short vector.
3. The cost is approximately $2^{0.292\beta}$ classical operations (Core-SVP model).

For 128-bit security, we need $\beta \geq 439$, which corresponds to ML-KEM-768 parameters.

9.2 Quantum Security

The best known quantum speedup for lattice attacks is Grover-like: $2^{0.265\beta}$ instead of $2^{0.292\beta}$. This is a modest improvement—unlike the exponential speedup Shor provides against RSA/ECC. Lattice schemes with 128-bit classical security provide approximately 117-bit quantum security, which is sufficient.

10 Common Implementation Pitfalls

1. **Timing leaks in rejection sampling**: The number of loop iterations in ML-DSA signing depends on the secret key. Use constant-time comparison and ensure the loop structure does not leak via timing. Our implementation always performs the norm check but masks the result.

2. **Incorrect NTT domain management**: NTT and inverse NTT must be applied in the right order. Adding polynomials in different domains (one in NTT form, one in coefficient form) is a common bug. Our type system distinguishes `Poly` from `NTTPoly` at the Go type level.

3. **Missing modular reduction**: Coefficients must be reduced modulo q after every arithmetic operation. A single missed reduction can cause subtle correctness failures that appear only for specific inputs.

4. **Endianness**: NIST test vectors use little-endian byte encoding for polynomials. Mixing up endianness produces code that passes self-consistency tests but fails against reference implementations.

11 Conclusion

Lattice cryptography is not magic—it is linear algebra with noise. LWE says: linear equations are easy; noisy linear equations are hard. Ring-LWE says: we can exploit polynomial ring structure for efficiency without losing hardness. Module-LWE says: we can parameterize security by matrix dimension for flexibility.

The NIST standards (ML-KEM, ML-DSA) are specific instantiations with concrete parameters chosen for practical security. Pulsar extends them to threshold operations for decentralized signing. The NTT makes all of it fast.

This paper covered the theory. Our companion papers cover the practice: NTT with SIMD acceleration, EVM precompiles for on-chain verification, and the formal proofs that tie it all together.

References

[1] Regev, O. "On Lattices, Learning with Errors, Random Linear Codes, and Cryptography." STOC, 2005.

[2] Lyubashevsky, V., Peikert, C., and Regev, O. "On Ideal Lattices and Learning with Errors over Rings." EUROCRYPT, 2010.

[3] Avanzi, R. et al. "CRYSTALS-Kyber: Algorithm Specifications and Supporting Documentation." NIST PQC Round 3, 2021.

[4] Ducas, L. et al. "CRYSTALS-Dilithium: Algorithm Specifications and Supporting Documentation." NIST PQC Round 3, 2021.

[5] NIST. "Module-Lattice-Based Key-Encapsulation Mechanism Standard (ML-KEM)." FIPS 203, 2024.

[6] NIST. "Module-Lattice-Based Digital Signature Standard (ML-DSA)." FIPS 204, 2024.

[7] Shor, P.W. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring." FOCS, 1994.

[8] Peikert, C. "A Decade of Lattice Cryptography." Foundations and Trends in Theoretical Computer Science, 2016.