Learning With Errors

Graeme Bates

April 30, 2025

Abstract

The goal of this paper is to give readers an intuitive understanding of the Learning With Errors problem (LWE). This paper gives a high level overview of the LWE and the associated hard lattice problems, and includes a simple implementation of the Ring-Learning With Errors (Ring-LWE) problem written in C++. However, this paper does not provide a comprehensive guide for constructing of a secure cryptosystem. Building a functional Ring-LWE system requires extensive analysis of a variety of design decisions, namely the choice of sampling function and relative magnitude of the error. Different approaches to sampling each have their merits, and balancing security against practicality is key. In general, increasing the noise, ie a larger error vector, increases security but introduces more errors, requiring error correcting or parity checks. Practical system design involves much thought on choice of prime modulus, error distribution, and error growth during encryption, especially when adding advanced features such as Fully Homomorphic Encryption.

1 Introduction

The **Learning With Errors** Problem is a modern cryptographic protocol under the umbrella of Lattice Based Cryptography. The problem was first proposed in 2005 by Oded Regev. Since then, variations, most notably the Ring Learning With Errors problem, have been used to create Public Key Cryptographic Protocols that are shown to be at least as hard as worst case Lattice Problems, ie NP hard problems (Problems that cannot be solved in polynomial time).

2 Motivation

The **Motivation** for Learning With Errors (LWE) along with lattice based cryptography in general, stems from the search of Quantum safe cryptographic systems. Problems such as the discrete logarithm and RSA are secure against traditional attacks but are vulnerable to Quantum algorithms. Researchers looking to a future where quantum computers could undermine the security of cyberspace have since proposed quantum safe protocols such

as LWE, with the Ring-LWE problem being a strong candidate for the future of data security.

Another major appeal of LWE-based schemes is their support for Fully Homomorphic Encryption (FHE), which allows arbitrary computation on encrypted data without requiring decryption. While FHE presents a breakthrough in secure computing, practical implementations still face challenges, including error accumulation and ciphertext expansion. Techniques such as bootstrapping aim, which computes 'partial' decryption steps to mitigate these issues.

3 Lattice Preliminaries

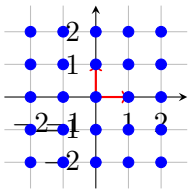
3.1 Lattices

Definition: A set $\mathcal{L}$ generated by a set of basis vectors $\mathcal{B} = \{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n : \mathbf{b}_i \in \mathbb{R}^m\}$, define a lattice:

$$\mathcal{L} = \{\sum_{i=1}^{n} k_i \mathbf{b}_i : k_i \in \mathbb{Z}, b_i \in \mathcal{B}\}$$

A lattice is the set of all integer linear combinations of a basis — forming a discrete additive subgroup of Euclidean space.

ex Given the standard basis for $\mathbb{R}^2$, the lattice would look like:



3.2 Basis for Lattice Complexity

A "good" basis is one that consists of short, nearly orthogonal vectors and makes lattice problems easier to solve, even in higher dimensions. The intuition behind a good basis, is it makes the lattice easy to "navigate", in the sense that one can pick a direction and move that way due to the near orthogonality of the basis. A "bad" basis, where the basis vectors are long and highly non-orthogonal, make Lattice problems extremely difficult, especially as the dimension increases. With long almost parallel vectors one takes a step forward and two steps back.

3.3 Hard Lattice Problems

The Security of LWE is based on the hardness of Lattice problems which LWE and its variations can be reduced to. A key lattice problem is the **Closest Vector Problem** (CVP), in which, given a point in $\mathbb{R}^n$, one must determine the nearest lattice point. The difficulty of CVP depends on the quality of the basis: a good basis makes it solvable efficiently, whereas a bad basis makes it computationally hard.

The CVP is particularly important because it serves as the foundation for proving the time complexity of many Lattice problems. CVP is an NP-hard problem through reduction to other known NP-hard problems. A known hard problem can be reduced to CVP, showing that CVP is at least as hard as a known NP-HardProblem, making CVP NP-hard. Example would be the Subset Sum Problem. It is easy to see why such a problem is hard. There is no systematic way of knowing when all lattice points near the origin have generated using a given "bad" basis, hence there is not even a slow way to try all possibilities as one could with RSA or other polynomial time problems.

The **Bounded Distance Decoding** BDD problem is a special case of CVP, and involves finding the closest lattice point to a given target vector, the target vector is guaranteed to be within a tolerance distance from the lattice. Although this problem sounds very similar to CVP, the guarantee of the target point being a given distance away from the lattice and the uniqueness of the resulting solution distinguish the problem. The hardness of Learning With Errors, is based on the reduction of LWE to BDD.

3.4 NTRU Cryptosystem

The NTRU encryption scheme is a public-key cryptographic scheme based on polynomial rings. Although the NTRU system varies from Learning With Errors in terms of key generation, a variant of the Learning With Errors uses polynomial rings to increase efficiency and has a public key system modeled off the NTRU system. For more details on NTRU system see appendix A.

4 Learning With Errors (LWE)

At its heart, the Learning With Errors problem involves solving a system of linear equations. Many efficient and sound methods, such as Gaussian elimination and row reduction, exist to solve such systems of linear equations, which is where the "error" in LWE comes from. Noise is added to the system so that one cannot explicitly solve the system. Rather than a system of linear equations, the goal of LWE is to solve for a secret vector, given a system of linear approximations. Using traditional methods like row reduction results in compounding error and will not yield the desired solution.

4.1 The Learning With Errors Problem

All arithmetic for LWE is done modulo q, q a large prime.

$$(\mathbb{Z}/\mathbb{Z}q)^n = \underbrace{(\mathbb{Z}/\mathbb{Z}q) \times (\mathbb{Z}/\mathbb{Z}q) \times \cdots \times (\mathbb{Z}/\mathbb{Z}q)}_{n \text{ times}}$$

will be notated $\mathbb{Z}_q^n$.

4.1.1 Key Generation

LWE is a public key encryption system. Alice creates a private key by selecting a random secret vector

$$\mathbf{s} = (s_1, s_2, s_3, \ldots, s_n) \in \mathbb{Z}_q^n$$

where each s_i is chosen uniformly from $\mathbb{Z}_q$ the integers modulo q. Once Alice has determined her secret vector, she generates an overdetermined system of linear equations using $\mathbf{s}$.

$$\begin{aligned} a_1^1 s_1 + a_2^1 s_2 + \cdots + a_n^1 s_n &\equiv x_1 \text{ mod q} \\ a_1^2 s_1 + a_2^2 s_2 + \cdots + a_n^2 s_n &\equiv x_2 \text{ mod q} \\ &\cdots \\ a_1^m s_1 + a_2^m s_2 \cdots + a_n^m s_n &\equiv x_m \text{ mod q} \end{aligned}$$

Which can be expressed in matrix form as $\mathbf{As} = \mathbf{x}$. Each entry a_j^k is chosen uniformly at random from $\mathbb{Z}_q$.
A small error vector:

$$\mathbf{e} = (e_1, e_2, \ldots, e_m) \in \mathbb{Z}^m$$

is generated, with each e_i being sampled from a discrete error distribution.
The noisy equations are then determined:

$$\mathbf{y} = As + e$$

Alice's public key is then: $(A, \mathbf{y})$
Alice's secret key: $\mathbf{s}$

4.1.2 Encoding

To encrypt a message, Bob proceeds as follows
Bob wants to encode the binary string:

$$\mathbf{p} = (p_1, p_2, p_3, ..., p_l)$$

For each bit p_i, Bob selects a random subset of equations from Alice's public key and adds them together to generate a new equation.
Each equation in Alice's public key is of the form:

$$a_1^t s_1 + a_2^t s_2 + \cdots + a_n^t s_n \equiv y_t \bmod \text{q}$$

Bob randomly selects several equations and sums them.

$$(a_1^i s_1 + a_2^i s_2 + \cdots + a_n^i s_n) + (a_1^j s_1 + a_2^j s_2 + \cdots + a_n^j s_n) + \cdots \equiv y_i + y_j + \ldots$$

Which we simplify to:

$$b_1^i s_1 + b_2^i s_2 + \cdots + b_n^i s_n \equiv m_i \bmod \text{q}$$

Where:

$\mathbf{b}_i = (b_1^i, b_2^i, \ldots, b_n^i)$ is the sum of randomly selected rows of A.

$\mathbf{m}_i = y_i + y_j + \ldots \bmod q$ is the sum of corresponding values of $\mathbf{y}$ for given rows of A

Then for each bit p_i, the cipher block $\mathbf{c} = c_1 c_2 \ldots c_l$

$$\begin{cases} c_i = m_i & p_i = 0 \\ c_i = m_i + \lfloor \frac{q}{2} \rceil & p_i = 1 \end{cases}$$

The shift of $\lfloor \frac{q}{2} \rceil$ will allow Alice to distinguish the bit despite the inherent noise in the system.
Bob constructs the cipher text:

$$\begin{aligned} b_1^1 s_1 + b_2^1 s_2 \cdots + b_n^1 s_n &\equiv c_1 \bmod \text{q} \\ b_1^2 s_1 + b_2^2 s_2 \cdots + b_n^2 s_n &\equiv c_2 \bmod \text{q} \\ &\cdots \\ b_1^l s_1 + b_2^l s_2 \cdots + b_n^l s_n &\equiv c_l \bmod \text{q} \end{aligned}$$

and sends $(\mathbf{B}, \mathbf{c})$ to Alice.
It is worth noting that the modulus $\mathbf{q}$ and size of error vector will dictate the number of equations Bob will choose, since too many error terms summed up can mess up the noisy decoding.

4.1.3 Decoding

After Bob sends his system of equations to Alice, Alice can use her secret key $\mathbf{s}$ to find the actual solution to each equation.

$$\mathbf{B} \cdot \mathbf{s} = \mathbf{w}$$

Then Alice takes $c_i - w_i \equiv d_i \bmod q$ to get original message,

$$\begin{cases} p_i = 0 & d_i \sim 0 \\ p_i = 1 & d_i \sim \frac{q}{2} \end{cases}$$

Since the noise is small, d_i will be near either 0 or $q/2$, enabling recovery of bit.

4.2 Drawbacks

Although the LWE system can be made extremely secure and is relatively simple to implement, the cost both computationally and in memory make it very impractical for usage. In order to encode a single bit, multiple calculations need to be made and then $n+1$ integers are needed to be stored for each bit, one integer for each of the n dimensions and finally an extra integer representing the bit + solution + error.

4.3 Ring-LWE

Due to its practical limitations, a variation on the LWE problem is commonly used to take advantage of the LWE security properties while improving efficiency. The Ring-LWE problem, similar to the NTRU system mentioned above, uses elements of a quotient ring rather than a typical lattice. The Ring-LWE can be reduced to a known hard problem, the approximate shortest independent vectors problem (SIVP), which is a variant of CVP mentioned above, making it a hard problem just like LWE. Thanks to the algebraic structure of the ring, both computational and memory efficiency are significantly improved.

The Scheme works as follows:

Given integers n and q, with q a large prime, define the quotient ring $\mathbb{Z}_q[x]/(x^n+1)$. "Small" elements of the ring are polynomials with coefficients << q.
Elements of the ring have the form $a_0 + a_1x + a_2x^2 + ...a_nx^{n-1} \bmod q$, to shorten notation, elements of the ring will be written as vectors $< a_0\ a_1\ a_2\ \dots\ a_{n-1}\ >$

1. A private key $s(x)$ is a random small element from the quotient ring.
2. A public key is represented by two polynomials $a(x), b(x)$. $a(x)$ is taken as a random element of the ring.
$$b(x) = a(x) \cdot s(x) + e(x)$$
Where $e(x)$ is a small ring element error term.
3. To Encrypt a bit string $b = b1, b2, b3, ...bl\ bi \in \{0,1\}$ of length $l < n$, firstly the bit string is converted into an element of the ring, $m(x) =< m1\ m2\ \dots\ ml\ >$ with $m_i = \begin{cases} 0 & b_i = 0 \\ q/2 & b_i = 1 \end{cases}$ A small random vector $r(x)$ is created as well as two small error

terms $e_1(x)$ and $e_2(x)$. The cipher text is represented by two ring elements, $c_1(x)$ and $c_2(x)$ with:

$$c_1(x) = a(x) \cdot r(x) + e_1(x)$$
$$c_2(x) = b(x) \cdot r(x) + e_2(x) + m(x)$$

4. To Decode one simply recognizes that $m(x) + \text{noise} = c_2(x) - s(x) \cdot c_1(x)$. Since the noise is made relatively small compared to q, rounding will recover the original message.

Here we see that an entire bit string can be encrypted using only two ring elements and efficient polynomial operations. This leads to significant improvements in both memory and computation compared to standard LWE.

Additionally, algorithms such as the Number Theoretic Transform (NTT) allow polynomial multiplication to be performed efficiently in the ring $\mathbb{Z}_q[x]/(x^n+1)$, making Ring-LWE suitable for practical cryptographic systems.

These optimizations have made Ring-LWE a leading candidate for post-quantum cryptography and forms the foundation for modern schemes such as Kyber, which is currently under consideration for standardization by the National Institute of Standards and Technology (NIST).

5 Example Implementation

The following section includes an example implementation of the RIng-LWE scheme. For full code and example usage please see `https://github.com/graemegilmourbates/ring-lwe`.
See sampling methods in appendix B.

The Following Type alias's are in use:

```
using Modulus = int;
using Dimension = int;
using Polynomial = std::vector<int>;
using CipherText = std::pair<Polynomial, Polynomial>;
using Key = Polynomial;
using PublicKey = std::pair<Polynomial, Polynomial>;
using BitVector = std::vector<char>;
```

5.1 Polynomial Operations

```
Polynomial mod_poly(const Polynomial& a, Modulus q){
```

```
  /* Handles positive and negative mods */
  Polynomial a_prime(a.size(), 0);
  int i=0;
  for(const auto& coef : a){
    a_prime[i] = ((coef % q)+q)%q;
    ++i;
  }
  return a_prime;
}

Polynomial add_poly(const Polynomial& a, const Polynomial& b, Modulus q){
  /**/
  int n = std::max(a.size(),b.size());
  Polynomial c(n,0);
  for(int i=0; i<n; ++i){
    if(i<a.size()) c[i] = c[i]+a[i];
    if(i<b.size()) c[i] = c[i]+b[i];
  }
  return mod_poly(c, q);
}

Polynomial subtract_poly(const Polynomial& a, const Polynomial& b, Modulus q){
  /**/
  int n = std::max(a.size(),b.size());
  Polynomial c(n,0);
  for(int i=0; i<n; ++i){
    if(i<a.size()) c[i] = c[i]+a[i];
    if(i<b.size()) c[i] = c[i]-b[i];
  }
  return mod_poly(c, q);
}

Polynomial multiply_poly(const Polynomial& a, const Polynomial& b, Modulus q){
  /* There exists a variation of the Fast Fourier Transform (FFT) that can greatly
   increase efficiency for polynomial ring multiplication that is not used here.*/
  int u = a.size();
  int w = b.size();
  int n = u+w-1;
  Polynomial c(n,0);
  for(int i=0; i<u; ++i){
    for(int j=0; j<w; ++j){
```

```
      c[i+j] += a[i]*b[j];
    }
  }
  return mod_poly(c, q);
}

Polynomial cyclic_mod(Polynomial a, Dimension n, Modulus q){
  /**/
  if(a.size() <= n) return mod_poly(a,q);
  // initialize zero polynomial
  Polynomial a_prime(n,0);
  int i=0, l, k;
  for(auto& coef : a){
    l = i%n;
    k = i/n;
    int sign = ((k%2==0) ? 1:-1);
    a_prime[l] = (a_prime[l]+coef*sign)%q;
    ++i;
  }
  return mod_poly(a_prime, q);
}
```

5.2 Key Generation

```
Key private_key_gen(Dimension n, Modulus q){
  /*
    our secret key s(x) is an element of the quotient ring Z_q[x]/(x^n+1)
    ie, has degree at most n-1
  */
  // small vector key, small ~ sqrt(q)/8
  return uniform_poly_sample(n,std::sqrt(q)/8);
}

PublicKey public_key_gen(Key s, Dimension n, Modulus q){
  /*
    a(x) is a random polynomial in the quotient ring.
    b(x) = a(x)*s(x)+e(x) mod q, where e(x) is a small error term
  */
  Polynomial a = uniform_poly_sample(n, q);
  Polynomial b = multiply_poly(a, s, q);
  double sigma = compute_sigma(n, q);
```

```
  Polynomial e = gaussian_error(n, sigma, q);
  b = add_poly(b, e, q);
  b = cyclic_mod(b, n, q);
  return std::make_tuple(a,b);
}
```

5.3 Encoding

```
CipherText encode(
  const PublicKey& p, const BitVector& msg, Dimension n, Modulus q
){
  /*
    b=b1,b2,b3,...bl bi in {0,1} (ie bit vector)
    m(x) = <m1 m2 m3 ... ml > mi=0 if bi=0 mi=q/2 if bi=1

    r(x) is small random vector
    c1(x) = a(x)*r(x)+e1(x)
    c2(x) = b(x)*r(x)+e2(x)+m(x)

    return (c1,c2)
  */
  Polynomial m(n,0);
  int i=0;
  // populate message polynomial... may want to add check that b.size()<n
  for(auto& bit : msg){
    if(bit=='1')m[i]=(q/2)+1;
    ++i;
  }
  double sigma = compute_sigma(n, q);
  // Get random small vectors
  Polynomial r = sample_ternary(n);
  Polynomial e1 = gaussian_error(n, sigma, q);
  Polynomial e2 = gaussian_error(n, sigma, q);
  Polynomial a, b, c1, c2;
  a = std::get<0>(p);
  b = std::get<1>(p);

  c1 = multiply_poly(a, r, q);
  c1 = add_poly(c1, e1, q);

  c2 = multiply_poly(b, r, q);
```

```
  c2 = add_poly(c2,e2,q);
  c2 = add_poly(c2,m,q);

  c1 = cyclic_mod(c1, n, q);
  c2 = cyclic_mod(c2, n, q);

  return std::make_tuple(c1, c2);
}
```

5.4 Decoding

```
Polynomial filter_noise(const Polynomial& m, Modulus q) {
  Polynomial clean(m.size(), 0);
  for (size_t i = 0; i < m.size(); ++i) {
    Modulus noisy = m[i] % q;
    clean[i] = (noisy >= q/4 && noisy <= 3*q/4) ? 1 : 0;
  }
  return clean;
}

Polynomial decode(const Key& s, const CipherText& c, Dimension n, Modulus q){
  /*
    c2(x)-s(x)c1(x)=m(x)+noise
  */
  Polynomial c1,c2,m;

  c1 = std::get<0>(c);
  c2 = std::get<1>(c);
  c1 = multiply_poly(c1, s, q);
  m = subtract_poly(c2, c1, q);
  m = cyclic_mod(m,n,q);
  m = filter_noise(m, q);
  return m;
}
```

References

[1] Oded Regev. *On Lattices, Learning with Errors, Random Linear Codes, and Cryptography.* In Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC), 2005, pp. 84–93.

[2] Chris Peikert. *A Decade of Lattice-Based Cryptography.* Foundations and Trends in Theoretical Computer Science, Vol. 10, No. 4, 2016, pp. 283–424.

[3] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. *On Ideal Lattices and Learning with Errors over Rings.* Journal of the ACM, Vol. 60, No. 6, 2013, Article 43.

[4] Oded Regev. *The Learning with Errors Problem.* Invited survey, available at `https://www.cims.nyu.edu/~regev/papers/lwesurvey.pdf`, 2010.

[5] Stephen Harrigan. *Lattice-Based Cryptography and the Learning with Errors Problem.* University of Ottawa, 2020. Undergraduate thesis.

A : NTRU protocol

A.0.1 Key Generation

Given a polynomial ring:

$$R = \mathbb{Z}[\mathbf{X}]/(f(\mathbf{X}))$$

where $f(\mathbf{X}) = \mathbf{X}^n - 1$ for n prime, or $f(\mathbf{X}) = \mathbf{X}^n + 1$ for n a power of 2.

Taking the quotient modulo some large odd number q:

$$R_q = R/qR$$

Bob generates two random polynomials $\mathbf{f}$ and $\mathbf{g}$ of degree at most $N-1$, with coefficients in $\{-1, 0, 1\}$. Bob ensures that $\mathbf{f}$ is invertible modulo both p and q, meaning there exist polynomials $\mathbf{f}_p$ and $\mathbf{f}_q$ such that:

$$\mathbf{f} \cdot \mathbf{f}_p \equiv 1 \pmod{p}, \quad \mathbf{f} \cdot \mathbf{f}_q \equiv 1 \pmod{q}$$

Bob computes his public key:

$$\mathbf{h} = p\mathbf{f}_q \cdot \mathbf{g} \mod q$$

Bob's private key consists of $\mathbf{f}$, $\mathbf{f}_q$, and $\mathbf{g}$.

A.0.2 Encryption and Decryption

When Alice wants to send Bob a message, she translates her message into a polynomial $\mathbf{m}$. The encryption and decryption process involves operations in the ring R_q, making use of Bob's private key for decryption.

B : Ring-LWE Sampling Methods

```
// Sample from a bounded discrete Gaussian (centered at 0, std_dev )
int sample_gaussian(std::mt19937& rng, double sigma, int bound) {
    std::normal_distribution<double> gaussian(0.0, sigma);
    while (true) {
        double val = gaussian(rng);
        int rounded = static_cast<int>(std::round(val));
        // Reject if outside bounds or with negligible probability
        if (std::abs(rounded) <= bound) {
            return rounded;
```

```
        }
    }
}

Polynomial gaussian_error(int n, double sigma, int q) {
    std::random_device rd;
    std::mt19937 rng(rd());
    Polynomial error_poly;
    int bound = std::min(static_cast<int>(6 * sigma), q / 2);

    for (int i = 0; i < n; ++i) {
        int coeff = sample_gaussian(rng, sigma, bound);
        coeff = (coeff % q + q) % q;
        error_poly.push_back(coeff);
    }
    return error_poly;
}

Polynomial uniform_poly_sample(Dimension n, Modulus q){
  /**/
  Polynomial p;
  std::random_device rand; // seed
  std::mt19937 rnd(rand()); // twister seed eng with seeded rand()
  std::uniform_int_distribution<> distrib(0,q-1);
  for(int i=0; i<n; ++i){
    // add uninformly at random unsigned int from [1,q]
    p.push_back(distrib(rnd));
  }
  return p;
}

Polynomial sample_ternary(Dimension n){
  Polynomial p;
  std::random_device rd;
  std::mt19937 gen(rd());
  std::uniform_int_distribution<> distrib(-1, 1);
  for(int i = 0; i < n; ++i){
    p.push_back(distrib(gen));
  }
  return p;
}
```

```
double compute_sigma(Dimension n, Modulus q, double bound, double margin){
    //return q / (tail_bound * noise_margin * std::sqrt(n));
    return std::sqrt(q)/2;
}
```