Lattice-based Cryptography: ring-LWE

Jackson Walters

January 30, 2025

Outline

- Goal of the talk

Outline

- Goal of the talk
- Abbreviated history of cryptography

Outline

- Goal of the talk
- Abbreviated history of cryptography
- Relevant hard problems on lattices

Outline

- Goal of the talk
- Abbreviated history of cryptography
- Relevant hard problems on lattices
- What is homomorphic encryption?
- Implementation of ring-LWE in Rust

Outline

- Goal of the talk
- Abbreviated history of cryptography
- Relevant hard problems on lattices
- What is homomorphic encryption?
- Implementation of ring-LWE in Rust
- FIPS 203 standard and module-LWE

History of cryptography

- Scytale (Ancient Greece, 500 BCE) Uses a cylinder and a strip of parchment to create a transposition cipher.

History of cryptography

- Scytale (Ancient Greece, 500 BCE) Uses a cylinder and a strip of parchment to create a transposition cipher.
- Transposition Ciphers (Classical Era) Rearranging plaintext characters according to a fixed system; examples include the Caesar cipher and later polyalphabetic ciphers like Vigenère.

History of cryptography

- Scytale (Ancient Greece, 500 BCE) Uses a cylinder and a strip of parchment to create a transposition cipher.
- Transposition Ciphers (Classical Era) Rearranging plaintext characters according to a fixed system; examples include the Caesar cipher and later polyalphabetic ciphers like Vigenère.
- Data Encryption Standard (DES, 1970s): Symmetric-key algorithm developed by IBM, adopted as a US std. in 1977. Block cipher using a 56-bit key, later replaced due to vulnerability to brute-force attacks.

History of cryptography

- Scytale (Ancient Greece, 500 BCE) Uses a cylinder and a strip of parchment to create a transposition cipher.
- Transposition Ciphers (Classical Era) Rearranging plaintext characters according to a fixed system; examples include the Caesar cipher and later polyalphabetic ciphers like Vigenère.
- Data Encryption Standard (DES, 1970s): Symmetric-key algorithm developed by IBM, adopted as a US std. in 1977. Block cipher using a 56-bit key, later replaced due to vulnerability to brute-force attacks.
- Advanced Encryption Standard (AES, 2001) Successor to DES, chosen through a public competition. Symmetric cipher using Rijndael algorithm, widely used today for secure communications.

History of cryptography

- Scytale (Ancient Greece, 500 BCE) Uses a cylinder and a strip of parchment to create a transposition cipher.
- Transposition Ciphers (Classical Era) Rearranging plaintext characters according to a fixed system; examples include the Caesar cipher and later polyalphabetic ciphers like Vigenère.
- Data Encryption Standard (DES, 1970s): Symmetric-key algorithm developed by IBM, adopted as a US std. in 1977. Block cipher using a 56-bit key, later replaced due to vulnerability to brute-force attacks.
- Advanced Encryption Standard (AES, 2001) Successor to DES, chosen through a public competition. Symmetric cipher using Rijndael algorithm, widely used today for secure communications.
- RSA & ECC (1970s–1980s)
  RSA: First widely used public-key system based on the difficulty of factoring large numbers.
  ECDLP: Public-key system relying on f.g. abelian groups of elliptic curves. Similar security with smaller keys.

Shor's algorithm to break RSA

- In order to decrypt ciphertext $C = M^e \mod N$ quickly, we need to find the modular inverse of e, where $N = p \cdot q$ is a large semiprime.

Shor's algorithm to break RSA

- In order to decrypt ciphertext $C = M^e \mod N$ quickly, we need to find the modular inverse of e, where $N = p \cdot q$ is a large semiprime.
- We are working in the group $(\mathbb{Z}/N\mathbb{Z})^\times$, the multiplicative group of integers modulo N.

Shor's algorithm to break RSA

- In order to decrypt ciphertext $C = M^e \mod N$ quickly, we need to find the modular inverse of e, where $N = p \cdot q$ is a large semiprime.
- We are working in the group $(\mathbb{Z}/N\mathbb{Z})^\times$, the multiplicative group of integers modulo N.
- Note that the order of this group is given by the number of elements coprime to N, which is just $\varphi(N) = (q-1)(p-1)$.

Shor's algorithm to break RSA

- In order to decrypt ciphertext $C = M^e \mod N$ quickly, we need to find the modular inverse of e, where $N = p \cdot q$ is a large semiprime.
- We are working in the group $(\mathbb{Z}/N\mathbb{Z})^\times$, the multiplicative group of integers modulo N.
- Note that the order of this group is given by the number of elements coprime to N, which is just $\varphi(N) = (q-1)(p-1)$.
- Once $\varphi(N)$ is known, one can use the extended Euclidean algorithm to compute the modular inverse d of e, i.e. $d \cdot e \equiv 1 \mod N$. Then $C^d \equiv M^{de} \equiv M^1 \equiv M \mod N$.

Shor's algorithm: period to integer factors

- Shor's algorithm is a quantum algorithm which can be used to factor integers, and hence break RSA given enough qubits.
- The essential component of Shor's algorithm is finding the multiplicative order of an integer modulo N.

Shor's algorithm: period to integer factors

- Shor's algorithm is a quantum algorithm which can be used to factor integers, and hence break RSA given enough qubits.
- The essential component of Shor's algorithm is finding the multiplicative order of an integer modulo N.
- For if we have such an a such that $a^r \equiv 1 \mod N$, then we can write $a^r - 1 \equiv 0 \mod N$.

Shor's algorithm: period to integer factors

- Shor's algorithm is a quantum algorithm which can be used to factor integers, and hence break RSA given enough qubits.
- The essential component of Shor's algorithm is finding the multiplicative order of an integer modulo N.
- For if we have such an a such that $a^r \equiv 1 \mod N$, then we can write $a^r - 1 \equiv 0 \mod N$.
- As long as r is even (and it is with enough probability, so if not we just try again), we can write $(a^{r/2} - 1)(a^{r/2} + 1) \equiv 0 \mod N$.

Shor's algorithm: period to integer factors

- Shor's algorithm is a quantum algorithm which can be used to factor integers, and hence break RSA given enough qubits.
- The essential component of Shor's algorithm is finding the multiplicative order of an integer modulo N.
- For if we have such an a such that $a^r \equiv 1 \mod N$, then we can write $a^r - 1 \equiv 0 \mod N$.
- As long as r is even (and it is with enough probability, so if not we just try again), we can write $(a^{r/2} - 1)(a^{r/2} + 1) \equiv 0 \mod N$.
- This means we can extract a factor via $\gcd(a^{r/2} - 1, N)$ or $\gcd(a^{r/2} + 1, N)$ since we know $N | (a^{r/2} - 1)(a^{r/2} + 1)$.

Shor's algorithm in a nutshell

Shor's algorithm proceeds essentially in four steps:

- create a uniform superposition $2^{-N/2}\sum_{k=0}^{N-1}|k\rangle$ using Hadamard gates
- apply modular exponential gates $U|k\rangle = |a^k \mod N\rangle$

Shor's algorithm in a nutshell

Shor's algorithm proceeds essentially in four steps:

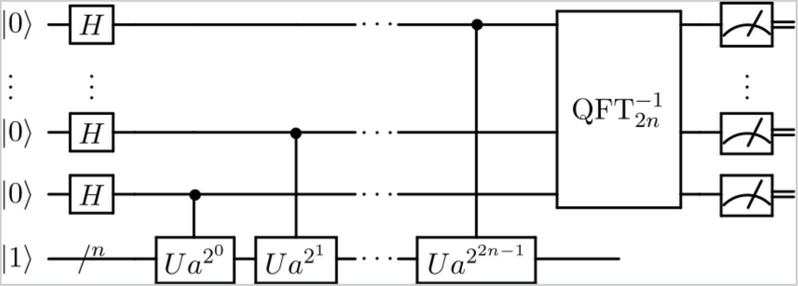
- create a uniform superposition $2^{-N/2}\sum_{k=0}^{N-1}|k\rangle$ using Hadamard gates
- apply modular exponential gates $U|k\rangle = |a^k \mod N\rangle$
- use the quantum Fourier transform (QFT) to perform phase estimation and extract period r

Shor's algorithm in a nutshell

Shor's algorithm proceeds essentially in four steps:

- create a uniform superposition $2^{-N/2} \sum_{k=0}^{N-1} |k\rangle$ using Hadamard gates
- apply modular exponential gates $U|k\rangle = |a^k \mod N\rangle$
- use the quantum Fourier transform (QFT) to perform phase estimation and extract period r
- use classical continued fractions to extract the actual period

Shor's algorithm: phase estimation circuit



Quantum subroutine in Shor's algorithm

Motivation for Modern Cryptography

For modern cryptography, we aim to find problems that are impractical or ideally impossible to solve, even on a quantum computer.

The "learning with errors" (LWE) problem, along with its ring-LWE variant, is an example of such problems. It involves distinguishing between two distributions:

- A set of random linear equations perturbed by a small error (noise)
- A truly uniform random distribution

The Learning with Errors Problem

Given $(\mathbf{A}, \mathbf{b}) \in \mathbb{Z}_{\mathbf{q}}^{\mathbf{n} \times \mathbf{m}} \times \mathbb{Z}_{\mathbf{q}}^{\mathbf{m}}$ where $\mathbf{b} = \mathbf{A}\mathbf{s} + \mathbf{e} \mod \mathbf{q}$:

- $\mathbf{A}$ is a known random matrix
- $\mathbf{s}$ is a secret vector
- $\mathbf{e}$ is a noise vector sampled from a narrow error distribution
- q is a large modulus

The goal is to recover the secret $\mathbf{s}$ or distinguish $(\mathbf{A}, \mathbf{b})$ from uniformly random samples. We'd like to know that instances of this problem are hard in the average case. One way to do this is by proving a reduction, that solving implies you've solved a "hard" problem in computer science. The two relevant reductions are:

Hardness of LWE

We would like to ensure that instances of LWE are hard in the average case. This is done by proving reductions from well-known hard problems in computer science:

- **GapSVP**: Shortest vector problem with a gap
- **SIVP**: Shortest independent vector problem

These lattice problems are:

- NP-hard in their exact versions
- Computationally hard in their approximate versions

The Ring-LWE Problem

The variant we focus on is ring-LWE, where the lattices are number-theoretic and derived from ideals in certain polynomial rings:

$$R_q = \mathbb{Z}_q[x]/(f(x))$$

where $f(x)$ is typically a polynomial like $x^n - 1$. However, this choice is insecure. Instead, we use:

$$f(x) = x^n + 1$$

where n is typically a power of two. This is the "anti-cyclotomic" ring of integers. The hard problem becomes:

- **Ideal** – **SVP**: Shortest vector problem for ideal lattices

Lattice Structure of the Ring

Consider the ring $R = \mathbb{Z}[x]/(x^n + 1)$:

- This is a $\mathbb{Z}$-module of rank n.
- Any element $a(x) \in R$ can be written as:

$$a(x) = \sum_{i=0}^{n-1} c_i x^i$$

- The vector $(c_i)_{i=0}^{n-1}$ is the coefficient vector, making $R \cong \mathbb{Z}^n$.

Connection Between Ring-LWE and Ideal Lattices

- $(x^n + 1)$ is an ideal, and elements $e(x) \in R$ map to elements in this lattice.
- The error $e(x)$ corresponds to a short vector (in the Euclidean norm) in the lattice, representing a small perturbation.

Ideal lattices inherit the ring's structure, such as multiplication by polynomials.

Reduction from Ring-LWE to Ideal-SVP

Solving ring-LWE allows efficient recovery of short vectors in ideal lattices:

- Ring-LWE enables recovery of the secret $s(x)$, which corresponds to information about the underlying lattice structure.
- Decoding the noisy lattice point perturbed by $e(x)$ reveals a short vector.

The reduction shows that solving ring-LWE allows decoding perturbed lattice points for any ideal lattice in R, solving Ideal-SVP in the process.

ring-LWE Overview

Our goal is to introduce the simplest possible implementation of the ring-LWE encryption scheme.

- Choose a moderately large prime p and large n
- n should be a power of two and 512 or above
- Example: $p = 3329$

Let

$$R_p := \mathbb{F}_p[x]/(x^n + 1)$$

This is a finite ring with p^n elements. It is not a finite field, as $x^n + 1$ factors modulo p (though it is irreducible over $\mathbb{Z}$). The elements are:

$$R_p = \{a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_1 x + a_0 : a_i \in \mathbb{F}_p\}$$

Public Setup

The public setup involves the following:

1. A prime p and dimension n resulting in R_p
2. A moderately large integer $k \in \mathbb{Z}$
3. A notion of "small", as applied to elements of R_p.
    - These will be "ternary polynomials" with coefficients in $\{-1, 0, +1\}$

Key Generation: Private/Public Keypair

Bob creates a private/public keypair as follows:

1. Bob selects a small random element s of R_p
2. Bob selects a small random element e_1 of R_p
3. Bob defines $(a, b = as + e_1) \in R_p \times R_p$

- The element e_1 can be discarded
- Bob keeps s as his secret key
- Bob makes (a, b) public as his public key

Encryption by Alice

Alice encrypts a message m as follows:

1. Select a small random $r \in R_p$ (ephemeral key)
2. Select small random $e_2, e_3 \in R_p$
3. Define $v = ar + e_2$, $w = br + e_3 + km$

- Alice may discard k, e_2, and e_3
- The ciphertext is (v, w), which is sent to Bob

Decryption by Bob

Bob decrypts the message:

1. Compute $x = w - vs$
2. Round x to the nearest multiple of k
3. The result should be an integer; divide it by k to reveal the message m

Homomorphic Encryption

Homomorphic encryption allows computations to be performed on encrypted data without needing to decrypt it first. This enables privacy-preserving computations.

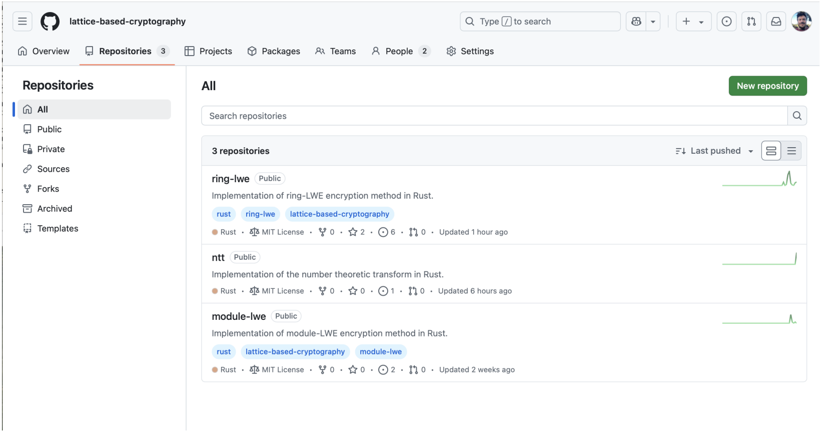
- **Encryption**: The data is encrypted using a homomorphic encryption scheme.
- **Computation**: Operations such as addition or multiplication are performed directly on the encrypted data.
- **Decryption**: The result of the computation is decrypted to reveal the final outcome.

Example: If $E(x)$ represents the encryption of data x, and $\oplus$ denotes an operation (like addition or multiplication):

$$E(x + y) = E(x) \oplus E(y) \quad \text{or} \quad E(x \times y) = E(x) \otimes E(y)$$

Homomorphic encryption can be used in cloud computing, privacy-preserving machine learning, and secure multi-party computations.

https://github.com/lattice-based-cryptography



ring-LWE: Library functions

```
use polynomial_ring::Polynomial;
use rand_distr::{Uniform, Normal, Distribution};
use ntt::polymul_ntt;
use rand::SeedableRng;
use rand::rngs::StdRng;

...

impl Default for Parameters {
    fn default() -> Self {
        let n = 16;
        let q = 65536;
        let t = 512;
        let mut poly_vec = vec![0i64;n+1];
        poly_vec[0] = 1;
        poly_vec[n] = 1;
        let f = Polynomial::new(poly_vec);
        Parameters { n, q, t, f }
    }
}
```

Listing 1: lib.rs

Number Theoretic Transform (NTT)

The Number Theoretic Transform (NTT) is a variant of the Fast Fourier Transform (FFT) used in modular arithmetic, often for computations in finite fields.

$$X_k = \sum_{n=0}^{N-1} x_n \cdot \omega_k^n \quad \text{for} \quad k = 0, 1, 2, \ldots, N-1$$

where:

- x_n is the input sequence of length N,
- X_k is the transformed sequence,
- $\omega = g^{(p-1)/N}$ is a primitive root of unity modulo a prime p,
- g is the generator for the field, and
- Use divide-and-conquer, recursively split for $\mathcal{O}(N \log(N))$

ring-LWE: ntt

```
// Forward transform using NTT, output bit-reversed
pub fn ntt(a: &[i64], omega: i64, n: usize, p: i64) -> Vec<i64> {
    let mut result = a.to_vec();
    let mut step = n/2;
        while step > 0 {
                let w_i  = mod_exp(omega, (n/(2*step)).try_into().unwrap(), p);
                for i in (0..n).step_by(2*step) {
                        let mut w = 1;
                        for j in 0..step {
                                let u = result[i+j];
                                let v = result[i+j+step];
                                result[i+j] = mod_add(u,v,p);
                                result[i+j+step] = mod_mul(mod_add(u,p-v,p),w,p)
                                     ;
                                w = mod_mul(w,w_i,p);
                        }
                }
                step/=2;
        }
        result
}
```

Listing 2: ntt

ring-LWE: polymul_fast

```
pub fn polymul_fast(x: &Polynomial<i64>, y: &Polynomial<i64>, q: i64, f: &
     Polynomial<i64>, root: i64) -> Polynomial<i64> {
    // Compute the degree and padded coefficients
    let n = 2 * (x.deg().unwrap() + 1);
    let x_pad = {
        let mut coeffs = x.coeffs().to_vec();
        coeffs.resize(n, 0);
        coeffs
    };
    let y_pad = {
        let mut coeffs = y.coeffs().to_vec();
        coeffs.resize(n, 0);
        coeffs
    };

    // Perform the polynomial multiplication
    let r_coeffs = polymul_ntt(&x_pad, &y_pad, n, q, root);

    // Construct the result polynomial and reduce modulo f
    let mut r = Polynomial::new(r_coeffs);
    r = polyrem(r,f);
    mod_coeffs(r, q)
}
```

Listing 3: polymul function

ring-LWE: Key generation

```
use polynomial_ring::Polynomial;
use ring_lwe::{Parameters, polymul, polyadd, polyinv, gen_binary_poly,
    gen_uniform_poly, gen_normal_poly};
use std::collections::HashMap;

pub fn keygen(params: &Parameters, seed: Option<u64>) -> ([Polynomial<i64>; 2],
    Polynomial<i64>) {

    //rename parameters
    let (n, q, f) = (params.n, params.q, &params.f);

    // Generate a public and secret key
    let sk = gen_binary_poly(n,seed);
    let a = gen_uniform_poly(n, q, seed);
    let e = gen_normal_poly(n, seed);
    let b = polyadd(&polymul(&polyinv(&a,q*q), &sk, q*q, &f), &polyinv(&e,q*q),
        q*q, &f);

    // Return public key (b, a) as an array and secret key (sk)
    ([b, a], sk)
}
```

Listing 4: keygen

ring-LWE: Encryption

```
use polynomial_ring::Polynomial;
use ring_lwe::{Parameters, mod_coeffs, polymul, polyadd, gen_binary_poly,
     gen_normal_poly};

pub fn encrypt(
    pk: &[Polynomial<i64>; 2],     // Public key (b, a)
    m: &Polynomial<i64>,          // Plaintext polynomial
    params: &Parameters,         //parameters (n,q,t,f)
    seed: Option<u64>            // Seed for random number generator
) -> (Polynomial<i64>, Polynomial<i64>) {
    let (n,q,t,f) = (params.n, params.q, params.t, &params.f);
    // Scale the plaintext polynomial. use floor(m*q/t) rather than floor (q/t)*
         m
    let scaled_m = mod_coeffs(m * q / t, q);

    // Generate random polynomials
    let e1 = gen_normal_poly(n, seed);
    let e2 = gen_normal_poly(n, seed);
    let u = gen_binary_poly(n, seed);

    // Compute ciphertext components
    let ct0 = polyadd(&polyadd(&polymul(&pk[0], &u, q*q, f), &e1, q*q, f),&
         scaled_m,q*q,f);
    let ct1 = polyadd(&polymul(&pk[1], &u, q*q, f), &e2, q*q, f);

    (ct0, ct1)
}
```

Listing 5: encrypt

ring-LWE: decryption

```
use polynomial_ring::Polynomial;
use ring_lwe::{Parameters, polymul, polyadd, nearest_int};

pub fn decrypt(
    sk: &Polynomial<i64>,     // Secret key
    ct: &[Polynomial<i64>; 2],        // Array of ciphertext polynomials
    params: &Parameters
) -> Polynomial<i64> {
    let (_n,q,t,f) = (params.n, params.q, params.t, &params.f);
        let scaled_pt = polyadd(&polymul(&ct[1], sk, q, f),&ct[0], q, f);
        let mut decrypted_coeffs = vec![];
        let mut s;
        for c in scaled_pt.coeffs().iter() {
                s = nearest_int(c*t,q);
                decrypted_coeffs.push(s.rem_euclid(t));
        }
    Polynomial::new(decrypted_coeffs)
}
```

Listing 6: decrypt

Homomorphic Encryption: Relinearization for Decryption

To decrypt the ciphertexts efficiently, relinearization is used. With s as secret key and ciphertext polynomials $c_i = (u_i, v_i)$, define

$$(c_0, c_1, c_2) := (v_0 * v_1, -(u_0 * v_1 + u_1 * v_0), u_0 * u_1)$$

To decrypt the ciphertext, one uses:

$$\lfloor \frac{c_0 + c_1 * s + c_2 * s * s}{\Delta^2} \rceil$$

where s is the secret key, and Δ is a scaling factor. Relinearization

simplifies the decryption process by reducing the number of ciphertext components, allowing for more efficient decryption.

ring-LWE: homomorphic product test

```
// Generate the keypair
let (pk, sk) = keygen(&params,seed);

// Encrypt plaintext messages
let u = encrypt(&pk, &m0_poly, &params, seed);
let v = encrypt(&pk, &m1_poly, &params, seed);

let plaintext_prod = &m0_poly * &m1_poly;
//compute product of encrypted data, using non-standard multiplication
let c0 = polymul(&u.0,&v.0,q*q,&f);
let u0v1 = &polymul(&u.0,&v.1,q*q,&f);
let u1v0 = &polymul(&u.1,&v.0,q*q,&f);
let c1 = polyadd(u0v1,u1v0,q*q,&f);
let c2 = polymul(&u.1,&v.1,q*q,&f);
let c = (c0, c1, c2);
//compute c0 + c1*s + c2*s*s
let c1_sk = &polymul(&c.1,&sk,q*q,&f);
let c2_sk_squared = &polymul(&polymul(&c.2,&sk,q*q,&f),&sk,q*q,&f);
let ciphertext_prod = polyadd(&polyadd(&c.0,c1_sk,q*q,&f),c2_sk_squared,
    q*q,&f);
//let delta = q / t, divide coeffs by 1 / delta^2
let delta = q / t;
let decrypted_prod = mod_coeffs(Polynomial::new(ciphertext_prod.coeffs()
    .iter().map(|&coeff| nearest_int(coeff,delta * delta) ).collect::<
    Vec<_>>()),t);

assert_eq!(plaintext_prod, decrypted_prod, "test failed: {} != {}",
    plaintext_prod, decrypted_prod);
```

Listing 7: decrypt

ring-LWE: benchmarking polymul_fast

```
jacksonwalters@jaxmacbookair ring-lwe % cargo run --bin benchmark

    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.08s
     Running `target/debug/benchmark`
Standard multiplication took: 13.75 mu s
Fast multiplication took: 47.959 mu s
Standard multiplication took: 4.241916ms
Fast multiplication took: 785.292 mu s
```

Listing 8: benchmark

FIPS 203

- NIST recently (Aug.) released their post-quantum cryptography standards
- CRYSTALS Kyber was selected for the KEM, forming the FIPS 203 standard

FIPS 203

- NIST recently (Aug.) released their post-quantum cryptography standards
- CRYSTALS Kyber was selected for the KEM, forming the FIPS 203 standard
- The core of this algorithm is module-LWE

FIPS 203

- NIST recently (Aug.) released their post-quantum cryptography standards
- CRYSTALS Kyber was selected for the KEM, forming the FIPS 203 standard
- The core of this algorithm is module-LWE
- See: `https://pq-crystals.org/kyber/` and `https://csrc.nist.gov/pubs/fips/203/final`

References I

Jackson Walters. *a brief introduction to quantum algorithms*.
https://jacksonwalters.com/blog/?p=1

Katherine Stange. *Ring-LWE notes*
https://math.colorado.edu/~kstange/teaching-resources/c

Jackson Walters, Thomas Silverman
https://github.com/lattice-based-cryptography