Module LWE

Thomas Silverman

January 23, 2025

Ring-LWE vs Module-LWE

Overview of Module-LWE

Implementation of Module-LWE in Rust

Ring-LWE vs Module-LWE

- Module-LWE is a generalization of ring-LWE, using the rank k free module R_q^k, where $R_q := \mathbb{Z}_q[x]/(x^n + 1)$ as in ring-LWE. The elements of R_q^k are k-tuples of polynomials in R_q. Ring-LWE is just the case $k = 1$.

Ring-LWE vs Module-LWE

- Module-LWE is a generalization of ring-LWE, using the rank k free module R_q^k, where $R_q := \mathbb{Z}_q[x]/(x^n + 1)$ as in ring-LWE. The elements of R_q^k are k-tuples of polynomials in R_q. Ring-LWE is just the case $k = 1$.
- Module-LWE works sort of like a hybrid between the original LWE scheme and ring-LWE, combining both matrix operations and polynomial multiplication.

Ring-LWE vs Module-LWE

Advantages of Module-LWE:

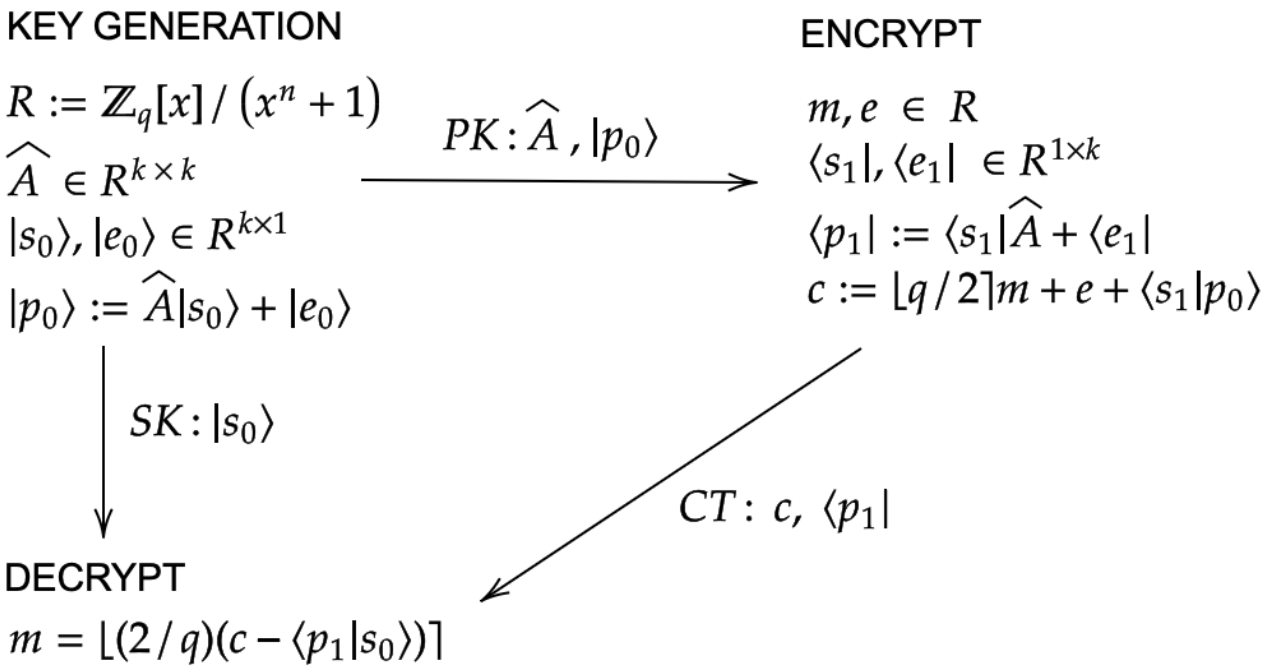
- Scalability: The main way to increase the security of ring-LWE is to increase the polynomial modulus degree n, but since this is always power of two there is no way to make a small increase in security if you are close to the necessary threshold. On the other hand, in module-LWE you can also increase the module rank k, which can be any integer.

Ring-LWE vs Module-LWE

Advantages of Module-LWE:

- Scalability: The main way to increase the security of ring-LWE is to increase the polynomial modulus degree n, but since this is always power of two there is no way to make a small increase in security if you are close to the necessary threshold. On the other hand, in module-LWE you can also increase the module rank k, which can be any integer.
- Parallelization: Many of the computations in module-LWE involves matrix operations, which can be easily parallelized.

Module-LWE Diagram



Public Setup

The public setup involves the following:

1. A prime q and modulus degree n resulting in the ring R_q.
2. Module rank k.
3. A notion of "small", as applied to elements of R_q^k.
   - These are k-tuples of ternary polynomials, with coefficients in $\{-1, 0, 1\}$.

Key Generation

Bob selects a private/public keypair as follows:

1. Bob selects a small random element $|s_0\rangle$ of $R_q^{k\times 1}$
2. Bob selects a small random element $|e_0\rangle$ of $R_q^{k\times 1}$
3. Bob selects a uniformly random matrix $\hat{A}$ in $R_q^{k\times k}$
4. Bob defines $|p_0\rangle := \hat{A}\,|s_0\rangle + |e_0\rangle$.

- The element $|e_0\rangle$ can be discarded
- Bob keeps $|s_0\rangle$ as his secret key
- Bob makes $(\hat{A}, |p_0\rangle)$ public as his public key

Encryption by Alice

Alice encrypts a message $m \in \mathbb{Z}_2[x]/(x^n + 1)$ as follows:

1. Alice selects a small random element $\langle s_1|$ of $R_q^{1\times k}$ (ephemeral key)
2. Alice selects small random $\langle e_1| \in R_q^{1\times k}$ and $e \in R_q$
3. Alice defines $\langle p_1| := \langle s_1| \hat{A} + \langle e_1|$ and $c := \lfloor q/2 \rceil\, m + e + \langle s_1|p_0\rangle$

- Alice may discard $\langle s_1|$, $\langle e_1|$, and e
- The ciphertext is $(c, \langle p_1|)$, which is sent to Bob

Decryption by Bob

Bob decrypts the message as follows:

1. Compute $x = c - \langle p_1 | s_0 \rangle$
2. Divide by $\lfloor q/2 \rceil$
3. Round the coefficients to the nearest integer modulo 2; the result is the message m

Decryption by Bob

Bob decrypts the message as follows:

1. Compute $x = c - \langle p_1 | s_0 \rangle$
2. Divide by $\lfloor q/2 \rceil$
3. Round the coefficients to the nearest integer modulo 2; the result is the message m

Correctness:

$$
\begin{aligned}
x &= c - \langle p_1 | s_0 \rangle \\
&= \left(\lfloor q/2 \rceil \, m + e + \langle s_1 | p_0 \rangle \right) - \left(\langle s_1 | \, \hat{A} + \langle e_1 | \right) | s_0 \rangle \\
&= \lfloor q/2 \rceil \, m + e + \langle s_1 | \, \hat{A} \, | s_0 \rangle + \langle s_1 | e_0 \rangle - \langle s_1 | \, \hat{A} \, | s_0 \rangle + \langle e_1 | s_0 \rangle \\
&= \lfloor q/2 \rceil \, m + \text{small polynomials},
\end{aligned}
$$

so dividing by $\lfloor q/2 \rceil$ and rounding coefficients to the nearest integer modulo 2 returns m

Module-LWE: Library functions

```rust
use polynomial_ring::Polynomial;
use rand_distr::{Uniform, Distribution};
use rand::SeedableRng;
use rand::rngs::StdRng;
pub mod ring_mod;
use ring_mod::{polyadd, polymul, gen_uniform_poly};

...

impl Default for Parameters {
    fn default() -> Self {
        let n = 32;
        let q = 59049;
        let k = 8;
        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, k, f }
    }
}
```

Listing 1: lib.rs

Module-LWE: Library functions

```
pub mod ring_mod;
use ring_mod::{polyadd, polymul, gen_uniform_poly};

...

pub fn mul_vec_simple(v0: &Vec<Polynomial<i64>>, v1: &Vec<Polynomial<i64>>,
    modulus: i64, poly_mod: &Polynomial<i64>) -> Polynomial<i64> {
        //take the dot product of two vectors of polynomials

        assert!(v0.len() == v1.len());
        // sizes need to be the same

        let mut result = Polynomial::new(vec![]);
        for i in 0..v0.len() {
                result = polyadd(&result, &polymul(&v0[i], &v1[i], modulus, &
                    poly_mod), modulus, &poly_mod);
        }
        result
}
```

Listing 2: Dot Product

Module-LWE: Library functions

```
pub fn mul_mat_vec_simple(m: &Vec<Vec<Polynomial<i64>>>, v: &Vec<Polynomial<i64
    >>, modulus: i64, poly_mod: &Polynomial<i64>) -> Vec<Polynomial<i64>> {
        //multiply a matrix by a vector of polynomials

        let mut result = vec![];
        for i in 0..m.len() {
                result.push(mul_vec_simple(&m[i], &v, modulus, &poly_mod));
        }
        result
}
```

Listing 3: Matrix Product

Module-LWE: Key Generation

```
use polynomial_ring::Polynomial;
use module_lwe::{Parameters, add_vec, mul_mat_vec_simple, gen_small_vector,
    gen_uniform_matrix};
use std::collections::HashMap;

pub fn keygen(
        params: &Parameters,
    seed: Option<u64> //random seed
) -> ((Vec<Vec<Polynomial<i64>>>, Vec<Polynomial<i64>>), Vec<Polynomial<i64>>) {
    let (n,q,k,f) = (params.n, params.q, params.k, &params.f);
    //Generate a public and secret key
    let a = gen_uniform_matrix(n, k, q, seed);
    let sk = gen_small_vector(n, k, seed);
    let e = gen_small_vector(n, k, seed);
    let t = add_vec(&mul_mat_vec_simple(&a, &sk, q, &f), &e, q, &f);

    //Return public key (a, t) and secret key (sk) as a 2-tuple
    ((a, t), sk)
}
```

Listing 4: keygen

Module-LWE: Encryption

```rust
use polynomial_ring::Polynomial;
use module_lwe::ring_mod::{polyadd,polysub,nearest_int};
use module_lwe::{Parameters, add_vec, mul_mat_vec_simple, transpose,
    mul_vec_simple, gen_small_vector};

pub fn encrypt(
    a: &Vec<Vec<Polynomial<i64>>>,
    t: &Vec<Polynomial<i64>>,
    m_b: Vec<i64>,
    params: &Parameters,
    seed: Option<u64>
) -> (Vec<Polynomial<i64>>, Polynomial<i64>) {
    //get parameters
    let (n, q, k, f) = (params.n, params.q, params.k, &params.f);
    //generate random ephermal keys
    let r = gen_small_vector(n, k, seed);
    let e1 = gen_small_vector(n, k, seed);
    let e2 = gen_small_vector(n, 1, seed)[0].clone(); // Single polynomial
    //compute nearest integer to q/2
    let half_q = nearest_int(q,2);
    // Convert binary message to polynomial
    let m = Polynomial::new(vec![half_q])*Polynomial::new(m_b);
    // Compute u = a^T * r + e_1 mod q
    let u = add_vec(&mul_mat_vec_simple(&transpose(a), &r, q, f), &e1, q, f);
    // Compute v = t * r + e_2 - m mod q
    let v = polysub(&polyadd(&mul_vec_simple(t, &r, q, &f), &e2, q, f), &m, q, f
        );

    (u, v)
}
```

Listing 5: encrypt

Module-LWE: Decryption

```
use polynomial_ring::Polynomial;
use module_lwe::{Parameters, mul_vec_simple};
use module_lwe::ring_mod::{polysub, nearest_int};

pub fn decrypt(
    sk: &Vec<Polynomial<i64>>,     //secret key
    q: i64,                        //ciphertext modulus
    f: &Polynomial<i64>,  //polynomial modulus
    u: &Vec<Polynomial<i64>>, //ciphertext vector
        v: &Polynomial<i64>              //ciphertext polynomial
) -> Vec<i64> {
        //Decrypt a ciphertext (u,v)
        //Returns a plaintext vector

        //Compute v-sk*u mod q
        let scaled_pt = polysub(&v, &mul_vec_simple(&sk, &u, q, &f), q, &f);
        let half_q = nearest_int(q,2);
        let mut decrypted_coeffs = vec![];
        let mut s;
        for c in scaled_pt.coeffs().iter() {
                s = nearest_int(*c,half_q) % 2;
                decrypted_coeffs.push(s);
        }
    decrypted_coeffs
}
```

Listing 6: decrypt

Module-LWE: Homomorphic Addition

```
pub fn test_hom_add() {
        let seed = None; //set the random seed
        let params = Parameters::default();
        let (n, q, f) = (params.n, params.q, &params.f);
        let mut m0 = vec![1, 0, 1];
        m0.resize(n, 0);
        let mut m1 = vec![0, 0, 1];
        m1.resize(n, 0);
        let mut plaintext_sum = vec![1, 0, 0];
        plaintext_sum.resize(n, 0);
        let (pk, sk) = keygen(&params,seed);
        // Encrypt plaintext messages
        let u = encrypt(&pk.0, &pk.1, m0, &params, seed);
        let v = encrypt(&pk.0, &pk.1, m1, &params, seed);
         // Compute sum of encrypted data
        let ciphertext_sum = (add_vec(&u.0,&v.0,q,f), polyadd(&u.1,&v.1,q,f));
        // Decrypt ciphertext sum u+v
        let mut decrypted_sum = decrypt(&sk, q, f, &ciphertext_sum.0, &
             ciphertext_sum.1);
        decrypted_sum.resize(n, 0);

        assert_eq!(decrypted_sum, plaintext_sum, "test failed: {:?} != {:?}",
             decrypted_sum, plaintext_sum);
}
```

Listing 7: homadd

References I


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


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