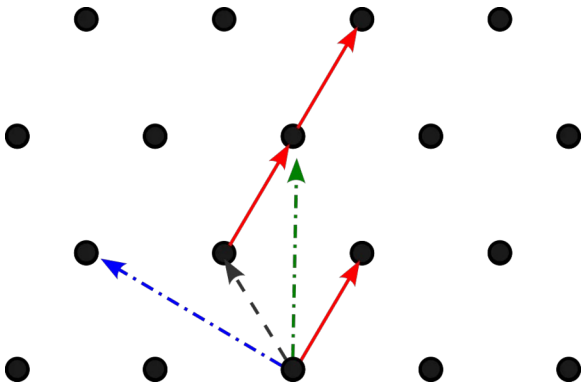
CS566

An Examination of Lattice-based Cryptography through the Kyber Algorithm



Ryan Christopher

Table of Contents:

Abstract

The Kyber Algorithm became one of the first post-quantum cryptographic methods to be standardized as ML-KEM by NIST as a result of their competition to find cryptographic methods that are resistant to quantum computers. This paper will provide a brief background to the type of cryptography that is used in Kyber, how it is useful in a situation where adversaries posses quantum machines, an examination of the pseudocode, and a comparison of how Kyber can perform compared to RSA as a public-key algorithm.

Lattice-based cryptography and how it differs from existing public key cryptography

Much of our public key cryptographic systems rely on one-way functions. These are mathematical functions which are easy to compute in one direction, yet very difficult to perform the inverse. In particular, prime factorization and solving the discrete logarithm problem are both one-way functions by which RSA and ECC – two dominant asymmetric-key encryption algorithms - generate public/private key pairs. In particular, these two methods for generating public/private keys were able to be computed with relative ease on most systems while maintaining computational infeasibility to attacks.

In 1994, Peter Shor, a mathematician and computer scientist developed an quantum algorithm that could easily compute prime factors for a given integer which was previously impossible for any classical computer [1]. This led cryptographers to pursue other potential forms of cryptography, and in the late 1990s lattice-based cryptography began to take form. Relying on the computational hardness of lattice problems, lattice-based cryptography employs the use of vectors to be used within cryptographic functions as there are concepts such as the "Shortest Vector Problem" that drastically

increase the computational hardness of the cryptographic methods as compared to the prior mentioned one-way functions.

Post-Quantum Cryptography and NIST

Shor's algorithm provided compelling evidence that quantum computers, once equipped with enough qubits and proper error correction, are particularly capable of solving prime factorization and discrete logarithm problems - the same two that make up much of our existing public key cryptographic systems. As such, there has been substantial effort placed into developing new cryptographic systems that are resistant to the capabilities of quantum computers. Post-Quantum Cryptography, or PQC, uses methods such as lattice-based cryptography which employs a different form of one-way functions.

The National Institute for Standards and Technology, NIST, deemed the need for standardization of post-quantum cryptography was necessary in 2017 [2], and announced their "Call for Proposals for Post-Quantum Cryptography Standardization." The competition involved mathematicians and cryptographers submitting their ideas for quantum-resistant cryptography, and if their proposed method was selected they had to implement improvements for the next round. What originated as 69 submissions became 4 standards announced in 2022 – one for key encapsulation and three for digitally signing.

Crystals, Kyber, ML-KEM, and FIPS 203

The Cryptographic Suite for Algebraic Lattices, known as CRYSTALS, was submitted to NIST's competition, and contains the Kyber algorithm for key encapsulation. Kyber combines the concept of vectors from lattice-based cryptography and obfuscates the vectors into a list of polynomial equations containing purposeful errors. These

purposeful errors make a shard secret extremely computationally hard for both classical machines and quantum machines, as the equations could be unsolvable without knowing the errors. This is the basis of the Learning With Errors (LWE) problem. Kyber generates private and public vector pairs that can share a secret key that is directly generated through mutual computations. By utilizing lattice-based cryptography with LWE, the algorithm is quantum resistant and serves as a viable means for securely generating a shared secret key.

After multiple rounds and revisions, Kyber was selected as the finalist for key encapsulation and was renamed to the Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM) Standard in the Federal Information Processing Standards publication series (FIPS). FIPS-203 states that at present, "ML-KEM is believed to be secure, even against adversaries who posses a quantum computer" [3].

Pseudocode for Kyber KEM

At its core, Kyber consists of three methods as a key-encapsulation mechanism: KeyGen, Encaps, and Decaps. Each step involves the use of internal algorithms for encryption, decryption, and formatting the polynomials used as vectors with a Number Theoretic Transform (NTT), however we will only take a look at the initial three methods for use of Kyber as a KEM. Once the shared key *K* is established, a method such as AES can be used for symmetric encryption.

Method 1: **KeyGen**

Input: none

Output: encapsulation key and decapsulation key

1. generate 32 random bytes for two variables *d* and *z*
2. call internal key generation with randomness (*d* and *z*) as input
3. return ek (encapsulation key), dk (decapsulation key)

Method 2: **Encaps**

Input: ek (encapsulation key)

Output: shared secret key (*K)* and ciphertext (*c)*

1. generate 32 random bytes for variable *m*
2. call internal encapsulation algorithm with randomness and encryption key as input (*m*, ek)
3. return shared secret key (*K*) and ciphertext (*c*)

Method 3: **Decaps**

Input: dk (decapsulation key) and ciphertext generated from Encapsulate method (*c*)

Output: shared secret key *K*

1. call internal decapsulate method with dk and *c*
2. return generated shared key *K*

Comparing the performance of Kyber to RSA

Kyber (later named ML-KEM) and RSA were implemented and are now maintained in Rust Crypto, a collection of “pure Rust implementations of cryptographic algorithms.” To compare the performance of Kyber compared to an commonly used public key algorithm like RSA, a text file containing random quotes was generated and both algorithms were assigned to a function.

Below is the code for Kyber:

```
fn kyber(message: String) -> String {
    let mut results: String = "".to_owned();
    // timestamps for total_time and stopwatch
    let total_time: Instant = Instant::now();
    let mut stopwatch: Instant = Instant::now();

    // instance of random number generator
    let mut rng: ThreadRng = rand::thread_rng();

    // encapsulate = public key, decapsulate = private key
    let (dk: DecapsulationKey<MlKem768Params>, ek: Encapsu…) = MlKem768::generate(&mut rng);
    results.push_str(string: &format!("Key generation time - {:?}\n", stopwatch.elapsed()));
    stopwatch = Instant::now();

    // encapsulate shared key to the holder of the decapsulation key, receive the shared secret 'k_send'
    // and the encapsulated form 'ct'
    let (ct: Array<u8, {unknown}>, k_send: Array<u8, UInt<UInt…) = ek.encapsulate(&mut rng).unwrap();
    results.push_str(string: &format!("Encapsulation time - {:?}\n", stopwatch.elapsed()));
    stopwatch = Instant::now();

    // decapsulate the shared key
    let k_recv: Array<u8, UInt<UInt<UInt<…, …>, …>, …>> = dk.decapsulate(en… &ct).unwrap();
    results.push_str(string: &format!("Decapsulation time - {:?}\n", stopwatch.elapsed()));

    // verify shared keys are identical
    assert_eq!(k_send, k_recv);
    stopwatch = Instant::now();

    // take symmetric key generated from kyber, pass as GenericArray
    let key: &GenericArray<u8, {unknown}> = Key::<Aes256Gcm>::from_slice(&k_send);

    // create cipher from key
    let cipher: AesGcm<Aes256, UInt<UInt<…, …>, …>> = Aes256Gcm::new(&key);
    results.push_str(string: &format!("Key and cipher generation time - {:?}\n", stopwatch.elapsed()));
    stopwatch = Instant::now();

    // initialize nonce/IV to be used in combination with encryption key
    let nonce: GenericArray<u8, UInt<UInt<…, …>, …>> = Aes256Gcm::generate_nonce(rng: &mut OsRng);

    // encrypt message
    let ciphertext: Vec<u8> = cipher.encrypt(&nonce, plaintext: message.as_bytes().as_ref()).unwrap();
    results.push_str(string: &format!("Encryption time - {:?}\n", stopwatch.elapsed()));
    stopwatch = Instant::now();

    // decrypt message
    let plaintext: Vec<u8> = cipher.decrypt(&nonce, ciphertext.as_ref()).unwrap();
    results.push_str(string: &format!("Decryption time - {:?}\n", stopwatch.elapsed()));

    // verify initial message and decrypted message are the same
    assert_eq!(&plaintext, message.as_bytes());
    results.push_str(string: &format!("Total time - {:?}\n", total_time.elapsed()));

    results
} fn kyber
```

Below is the code for RSA:

```rust
fn rsa(message: String) -> String {
    let mut results: String = "".to_owned();
    // timestamps for total_time and stopwatch
    let total_time: Instant = Instant::now();
    let mut stopwatch: Instant = Instant::now();

    // instance of random number generator
    let mut rng: ThreadRng = rand::thread_rng();

    // set bit size (therefore key size will be 256 bytes)
    let bits: usize = 2048;

    // generate private and public keys
    let priv_key: RsaPrivateKey = RsaPrivateKey::new(&mut rng, bit_size: bits).expect(msg: "failed to generate a key");
    let pub_key: RsaPublicKey = RsaPublicKey::from(&priv_key);
    results.push_str(string: &format!("Key generation time - {:?}\n", stopwatch.elapsed()));
    stopwatch = Instant::now();

    // encrypt message
    let data: &[u8] = message.as_bytes();
    let enc_data: Vec<u8> = pub_key.encrypt(&mut rng, padding: Pkcs1v15Encrypt, msg: &data[..]).expect(msg: "failed to encrypt");
    results.push_str(string: &format!("Encryption time - {:?}\n", stopwatch.elapsed()));
    stopwatch = Instant::now();

    // decrypt message
    let dec_data: Vec<u8> = priv_key.decrypt(padding: Pkcs1v15Encrypt, ciphertext: &enc_data).expect(msg: "failed to decrypt");
    results.push_str(string: &format!("Decryption time - {:?}\n", stopwatch.elapsed()));

    // verify initial message and decrypted message are the same
    assert_eq!(&data[..], &dec_data[..]);
    results.push_str(string: &format!("Total time - {:?}\n", total_time.elapsed()));

    results
} fn rsa
```

The algorithms were then used to take each line of the text file, generate keys, encrypt, and decrypt the message back to the original text. Then, the timestamps for each function were recorded and written to two text files (one for Kyber, one for RSA).

```rust
fn test_on_quotes(algorithm: &str){
    // get file name
    let mut stats_file_name: String = "./".to_owned();
    stats_file_name.push_str(&algorithm.to_string());
    stats_file_name.push_str(string: "_stats.txt");

    // open file, create if not found
    let mut stats_file: File = fs::OpenOptions::new().append(true).create(true).open(path: &stats_file_name).expect(msg: "Can not open file.");

    // for each line in quotes, call either kyber or rsa dependent on what the parameter was
    for line: &str in read_to_string(path: "./quotes.txt").unwrap().lines() {
        // show for progress
        println!("{}", line);
        if algorithm == "kyber" {
            stats_file.write_all(buf: line.as_bytes()).expect(msg: "Unable to read line.");
            stats_file.write_all(buf: "\n".as_bytes()).expect(msg: "Error writing new line");
            stats_file.write_all(buf: kyber(message: line.to_string()).as_bytes()).expect(msg: "Error on - kyber");
        }
        else if algorithm == "rsa" {
            stats_file.write_all(buf: line.as_bytes()).expect(msg: "Unable to read line.");
            stats_file.write_all(buf: "\n".as_bytes()).expect(msg: "Error writing new line");
            stats_file.write_all(buf: rsa(message: line.to_string()).as_bytes()).expect(msg: "Error on - rsa");
        }
    }
} fn test_on_quotes
```

Kyber test results:

```
If you want to achieve greatness stop asking for permission. ~Anonymous
Key generation time - 2.2915ms
Encapsulation time - 2.0103ms
Decapsulation time - 2.4412ms
Key and cipher generation time - 22.6µs
Encryption time - 31.5µs
Decryption time - 17.5µs
Total time - 6.827ms
Things work out best for those who make the best of how things work out. ~John Wooden
Key generation time - 1.9025ms
Encapsulation time - 1.9431ms
Decapsulation time - 2.407ms
Key and cipher generation time - 5.3µs
Encryption time - 22.6µs
Decryption time - 14.7µs
Total time - 6.3054ms
To live a creative life, we must lose our fear of being wrong. ~Anonymous
Key generation time - 2.0485ms
Encapsulation time - 1.9696ms
Decapsulation time - 2.4147ms
Key and cipher generation time - 4.4µs
Encryption time - 23.8µs
Decryption time - 14.3µs
Total time - 6.4832ms
If you are not willing to risk the usual you will have to settle for the ordinary. ~Jim Rohn
Key generation time - 1.901ms
Encapsulation time - 1.9283ms
Decapsulation time - 2.4168ms
Key and cipher generation time - 4.5µs
Encryption time - 20.6µs
Decryption time - 15.1µs
Total time - 6.2946ms
Trust because you are willing to accept the risk, not because it's safe or certain. ~Anonymous
Key generation time - 1.861ms
Encapsulation time - 1.9438ms
Decapsulation time - 2.4248ms
Key and cipher generation time - 4.1µs
Encryption time - 20.8µs
Decryption time - 13.9µs
Total time - 6.2781ms
Take up one idea. Make that one idea your life - think of it, dream of it, live on that idea. ~Swami Vivekananda
Key generation time - 1.8897ms
Encapsulation time - 1.9199ms
Decapsulation time - 2.4211ms
Key and cipher generation time - 4.4µs
Encryption time - 19.2µs
Decryption time - 16.9µs
Total time - 6.2777ms
All our dreams can come true if we have the courage to pursue them. ~Walt Disney
Key generation time - 1.8555ms
Encapsulation time - 1.899ms
Decapsulation time - 2.3949ms
Key and cipher generation time - 3.1µs
Encryption time - 14.6µs
Decryption time - 11.4µs
Total time - 6.1831ms
Good things come to people who wait, but better things come to those who go out and get them. ~Anonymous
Key generation time - 2ms
Encapsulation time - 1.9847ms
Decapsulation time - 2.463ms
Key and cipher generation time - 4.6µs
Encryption time - 26.4µs
Decryption time - 15.1µs
Total time - 6.5027ms
```

RSA test results:

```
If you want to achieve greatness stop asking for permission. ~Anonymous
Key generation time - 3.330196s
Encryption time - 2.6472ms
Decryption time - 21.5721ms
Total time - 3.3544278s
Things work out best for those who make the best of how things work out. ~John Wooden
Key generation time - 3.9448125s
Encryption time - 2.2734ms
Decryption time - 21.4396ms
Total time - 3.9685353s
To live a creative life, we must lose our fear of being wrong. ~Anonymous
Key generation time - 1.6714073s
Encryption time - 2.2117ms
Decryption time - 21.994ms
Total time - 1.6956206s
If you are not willing to risk the usual you will have to settle for the ordinary. ~Jim Rohn
Key generation time - 2.4325207s
Encryption time - 2.2599ms
Decryption time - 21.42ms
Total time - 2.456208s
Trust because you are willing to accept the risk, not because it's safe or certain. ~Anonymous
Key generation time - 2.5813378s
Encryption time - 2.2006ms
Decryption time - 21.7214ms
Total time - 2.6052692s
Take up one idea. Make that one idea your life - think of it, dream of it, live on that idea. ~Swami Vivekananda
Key generation time - 696.7755ms
Encryption time - 2.301ms
Decryption time - 21.2521ms
Total time - 720.3375ms
All our dreams can come true if we have the courage to pursue them. ~Walt Disney
Key generation time - 2.9944306s
Encryption time - 2.2484ms
Decryption time - 22.1246ms
Total time - 3.018812s
Good things come to people who wait, but better things come to those who go out and get them. ~Anonymous
Key generation time - 2.1839402s
Encryption time - 2.2125ms
Decryption time - 21.7474ms
Total time - 2.2079096s
```

The two resulting files yield promising results, as Kyber was far faster in key generation and total time required for the initial message to be received from decryption. The messages with Kyber required a total time of roughly 6.5ms, whereas the same messages using RSA required a total time of 720ms up to 3.3 seconds. It is worth noting that the key size for Kyber is larger than RSA, however the efficiency and security gained provides a compelling argument for the adoption of ML-KEM.

References

[1] Ugwuishiwu, C. H., et al. "An overview of quantum cryptography and shor's algorithm." *Int. J. Adv. Trends Comput. Sci. Eng* 9.5 (2020).

[2] "Call for Proposals - Post-Quantum Cryptography: CSRC." *NIST Computer Security Resource Center*, csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization/Call-for-Proposals. Accessed 21 Apr. 2025.

[3] *Module-Lattice-Based Key-Encapsulation Mechanism Standard*, 13 Aug. 2024, https://doi.org/10.6028/nist.fips.203.