Study & Analysis: Kyber

Gabriel C. Kinder – 234720

g234720@dac.unicamp.br

Abstract. *This paper analyzes the Kyber cryptographic system, explaining all of its functions and methods, presenting its underlying mathematical problem and an overview of the final algorithm. It also contains a discussion about its parameters selections as well as its efficiency, both memory wise and execution time wise.*

1. Prerequisites

1.1. Notation

For the representation of randomly generated numbers on our algorithm, we will utilize the following notation: $x = \beta^n$, where x receives n randomly generated bytes. A polynomial F represented as $\hat{F}$, denotes that the polynomial is represented on the NTT domain. We denote as *M[i]*, the i-th element of a vector M and as M^T its transpose. We represent by $a \,//\, b$ the concatenation of the elements a and b.

1.2. Arithmetic with polynomial rings

Kyber's main characteristic is the use of polynomial rings. Polynomial rings are a mathematical structure for modulo operations on polynomials. On Kyber, our polynomials will have a maximum degree of 255, and its coefficients will have a maximum value of 3328, those values are further discussed in section 5.

1.3. The Number Theoretic Transform

The main mathematical operation that will be done on Kyber is polynomial multiplications. For Kyber's parameters, polynomials have a degree of up to 255, which makes this sort of operation relatively expensive performance wise. For that reason Kyber chooses parameters so that a technique called Number Theoretic Transform (NTT) can be used to optimize this strategy.

The NTT is a generalization of the discrete Fourier Transform (DFT) and, in a simplistic way, reduces our polynomial of degree 255 to a vector of 128 polynomials of degree 1. With the correct parameters, this can speed up the process of polynomial multiplication considerably. Its important to note two properties of the NTT that makes this possible, the existence of an inverse of the NTT such that $NTT^{-1}(NTT(f)) = f$, and a distributive property such that $NTT(f * g) = NTT(f) * NTT(g)$, if we use those properties together we arrive at $f * g = NTT^{-1}(NTT(f) * NTT(g))$, which allows for the use of the NTT and its inverse to calculate our polynomial multiplications in a much quicker way. The result of such operations can be seen at section 4.2, allowing Kyber to be quicker than even modern non post-quantum algorithms.

2. Utility Functions

There are four utility functions used on Kyber's algorithm, a compressing function, an encoding function, a uniform sampling function (parse) and a binomial distribution sampling function (CBD).

2.1. Compress

The function $Compress_q(x, d)$, and its counterpart, $Compress_q^{-1}(x, d) =$

$\text{Decompress}_q(x, d)$ are used mainly to discard low-order bits from the ciphertext, where Decompress_q converts coefficient values that are 0 to 0 and values that are 1 to the closest integer of q/2 while Compress_q sends coefficients closer to q/2 to 1 and coefficients closer to 0 to 0. The defined functions are as follows:

$$Compress_q(x,\ d) = \lceil 2^d/q \cdot x \rfloor \ mod\, 2^d$$
$$Decompress_q(x,\ d) = \lceil (q/2^d) \rfloor \cdot x$$

When the Compress or Decompress function is used on a polynomial we apply the function to each of its coefficients individually. Similarly, when it is used on a matrix or vector, we apply the function to all of the matrix or vector elements individually.

2.2. Encode

For encoding, there are two data types that need to be serialized, byte arrays and polynomials. Byte arrays are serialized through their identity while polynomials follows the following Decode algorithm for decoding, and its inverse for encoding

Algorithm 1. Decode_l

Input: Byte array B such that $B=(b_0, b_1, \ldots, b_{255 \cdot l - 2}, b_{256 \cdot l - 1})$ where b_i represents the ith bit of the byte array

Output: Polynomial f

for i from 0 to 255:

$$f_i = \sum_{j=0}^{l-1} b_{i \cdot l + j} \cdot 2^j$$

return $f_0 + f_1 \cdot x + f_2 \cdot x^2 + \ldots + f_{255} \cdot x^{255}$

When the Encode or Decode function is used on a matrix or vector, we apply the function to each of the matrix or vector elements individually.

2.3. Parse

The Parse function is a deterministic algorithm to uniform sample elements in R_q which are statistically close to a uniformly random distribution. It receives a byte stream B as seed. We assume for this algorithm that q = 3329, a fixed parameter for all versions of Kyber.

Algorithm 2. Parse

Input: Byte stream B such that $B=(b_0, b_1, b_2, ...)$ where b_i represents the ith bit of the byte stream

Output: NTT representation of $a \in R_q$

$i = 0$
$j = 0$
while $j < n$:
  $d_1 = b_i + 256 \cdot (b_{i+1} \ mod 16)$
  $d_2 = \lfloor b_{i+1}/16 \rfloor + 16 \cdot b_{i+2}$
  if $d_1 < q$:
    $\hat{a} = d_1$
    $j = j + 1$
  if $d_2 < q$ and $j < n$:
    $\hat{a}_j = d_2$
    $j = j + 1$
  $i = i + 3$
return $\hat{a}$

2.4. CBD

The CBD function is used to generate "noise" in Kyber. It works as a sample from a centered binomial distribution β_η for $\eta = 2$ or 3. When a polynomial is sampled from our CBD we mean that each coefficient is sampled from the CBD individually. Similarly, when we say that a matrix or vector is sampled from CBD we mean that each element is individually sampled from the CBD.

Algorithm 3. CBD_η

Input: Bits array $\beta = \{b_0, b_1, ..., b_{512\eta-1}\}$

Output: Polynomial f

for i from 0 to 255:

$$a = \sum_{j=0}^{\eta-1} b_{2 \cdot i \cdot \eta + j}$$

$$b = \sum_{j=0}^{\eta-1} b_{2 \cdot i \cdot \eta + \eta + j}$$

$f_i = a - b$

return $f_0 + f_1 \cdot x + f_2 \cdot x^2 + ... + f_{255} \cdot x^{255}$

3. Kyber's Algorithm

Kyber is a key-encapsulation mechanism (KEM) created to be secure against both classical and quantum attacks and classified as IND-CCA2-secure. Kyber is constructed by two steps, first it is made an IND-CPA-secure public key encryption

scheme with a fixed length of 32 bytes, called Kyber CPAPKE, then, by using a tweaked Fujisaki-Okamoto (FO) transform [2], the Kyber CCAKEM is constructed, a key encapsulation method.

3.1. Kyber Parameter Sets

Kyber is divided into 3 different parameter sets with different security levels, as follows:

Table 1. Parameter sets for Kyber

	n	k	q	η_1	η_2	d_u	d_v	δ
Kyber512	256	2	3329	3	2	10	4	2^{-139}
Kyber768	256	3	3329	2	2	10	4	2^{-164}
Kyber1024	256	4	3329	2	2	11	5	2^{-174}

The reasoning behind each parameter choice is further described in section 5 of this document.

3.2. Overview

Before getting into more technical details a simpler and less detailed version of the algorithm can help understanding the idea of the algorithm, making it easier to understand the technical details later on. For the key generation we have the following formula:

$A \cdot s + e = t$ Where A is a k x k matrix and s, e and t are vectors of k dimensions and all of their elements are polynomials. A, s and e are generated randomly while t is calculated from the former three. We will use the pair (A, t) as our public key and s as our private key, while e is only an error to make the underlying mathematical problem hard to solve. Then for encryption, the message m will be converted to a polynomial representation and have all of its coefficients multiplied by the rounding of q/2 (1665). Utilizing the public key we encrypt the message as follows:

$$u = A^T \cdot r + e_1$$
$$v = t^T \cdot r + e_2 + m$$

Where m is our encoded and multiplied message, and e_1, e_2 and r are generated randomly. The pair (u, v) is our ciphertext c.

Finally, for decryption, we can use our private key s, together with the ciphertext to make the following conclusion:

$$d = v - s^T \cdot u = t^T \cdot r + e_2 + m - s^T \cdot (A^T \cdot r + e_1) \Rightarrow$$
$$\Rightarrow d = e^T \cdot r + e_2 + m - s^T \cdot e_1$$

Since all of our randomly generated terms are smaller than m, which is only being added with those terms, we can round the coefficients of each polynomial variable to whatever they are closest, 0 or q/2, and then set them back to either 0 or

1, obtaining our original message back.

3.3. Kyber CPAPKE

The Kyber CPAPKE is the algorithm described in the overview section. It's full technical details make an IND-CPA-secure algorithm, sending a message with a fixed size of 32 bytes. It is composed of 3 algorithms, a key generator, an encryption method and a decryption method called CPAPKE.KeyGen(), CPAPKE.Enc(pk, m, r) and CPAPKE.Dec(sk, c), respectively. The algorithm for the key generation is as follows:

Algorithm 4. CPAPKE.KeyGen()

Output: Secret key sk
Output: Public key pk
$d \leftarrow \beta^{32}$
$(\rho, \sigma) = G(d)$
$N = 0$
for i from 0 to $k - 1$:
  for j from 0 to $k - 1$:
    $\hat{A}[i][j] = Parse(XOF(\rho, j, i))$
for i from 0 to $k - 1$:
  $s[i] = CBD_{\eta_1}(PRF(\sigma, N))$
  $N = N + 1$
for i from 0 to $k - 1$:
  $e[i] = CBD_{\eta_1}(PRF(\sigma, N))$
  $N = N + 1$
$\hat{s} = NTT(s)$
$\hat{e} = NTT(e)$
$\hat{t} = \hat{A} \circ \hat{s} + \hat{e}$
$pk = (Encode_{12}(\hat{t} \bmod q) \parallel \rho)$
$sk = Encode_{12}(\hat{s} \bmod q)$
return (pk, sk)

What this algorithm does is generate A, s and e, and calculate t, returning the public key and private key as an output. The main difference from our key generation on the overview is that, instead of using t and A as a public key pair, we keep the NTT encoded value of t and the seed to generate A. This lowers the size of the public key considerably, since the matrix A has size k x k. We also keep the NTT encoded value of s instead of s, since further on, for the decryption process, we will use the NTT value of s for polynomial multiplication. The seeds for the Parse and CBD functions are generated through an extendable output function (XOF) and pseudo-random function (PRF) respectively. Those are better defined in section 3.5 on this document.

Next is the full technical details of the encryption algorithm for the CPAPKE. The main differences we can see here is the generation of the matrix A already being transposed, this saves some time since the multiplication afterwards would require its transposition. We can also see the usage of the Decompress function to prepare the message and the Compress function to make the ciphertext smaller.

Algorithm 5. CPAPKE.Enc(pk, m, r)

Input: Public key pk
Input: Message m
Input: Random coins r
Output: Ciphertext c

$N = 0$
$(t', \rho) = pk$
$\hat{t} = Decode_{12}(t')$
for i from 0 to $k - 1$:
  for j from 0 to $k - 1$:
    $\hat{A}^T[i][j] = Parse(XOF(\rho, i, j))$
for i from 0 to $k - 1$:
  $r[i] = CBD_{\eta_1}(PRF(r, N))$
  $N = N + 1$
for i from 0 to $k - 1$:
  $e_1[i] = CBD_{\eta_2}(PRF(r, N))$
  $N = N + 1$
$e_2 = CBD_{\eta_2}(PRF(r, N))$
$\hat{r} = NTT(r)$
$u = NTT^{-1}(\hat{A}^T \circ \hat{r}) + e_1$
$v = NTT^{-1}(\hat{t}^T \circ \hat{r}) + e_2 + Decompress_q(Decode_1(m), 1)$
$c_1 = Encode_{d_u}(Compress_q(u, d_u))$
$c_2 = Encode_{d_v}(Compress_q(v, d_v))$
return $c = (c_1 \| c_2)$

The final algorithm in the CPAPKE is the decryption algorithm. It is a much simpler algorithm than the other two.

Algorithm 6. Kyber.CPAPKE.Dec(sk, c)

Input: Secret key sk
Input: Ciphertext c
Output: Message m
$(c1, c2) = c$
$u = Decompress_q(Decode_{d_u}(c_1), d_u)$
$v = Decompress_q(Decode_{d_v}(c_2), d_v)$
$\hat{s} = Decode_{12}(sk)$
$m = Encode_1(Compress_q(v - NTT^{-1}(\hat{s}^T \circ NTT(u)), 1))$
return m

We simply calculate as explained on the overview, and use the Compress function to round the coefficients of the message m to their proper value, after properly decoding and decompressing the contents on the ciphertext.

3.4. Kyber CCAKEM

The Kyber CCAKEM is the final algorithm, being classified as an IND-CCA2 key encapsulation method. It is created after applying the FO transform to the CPAPKE algorithm. It is composed of 3 algorithms, a key generator, an encapsulation method and a decapsulation method called CCAKEM.KeyGen(), CCAKEM.Enc(pk) and CPAKEM.Dec(c, sk), respectively. The algorithm for the key generation is as follows:

Algorithm 7. Kyber.CCAKEM.KeyGen()

Output: public key pk
Output: secret key sk
$z \leftarrow \beta^{32}$
$(pk, sk') = Kyber.CPAPKE.KeyGen()$
$sk = (sk' \| pk \| H(pk) \| z)$
return (pk, sk)

The key generator for the CCAKEM simply adds more data to the secret key, like the hash of the public key and 32 random bytes to be used later on the algorithm.

Algorithm 8. Kyber.CCAKEM.Enc()

Input: Public key pk
Output: Ciphertext c
Output: Shared Key K
$m \leftarrow \beta^{32}$
$m = H(m)$
$(\bar{K}, r) = G(M \parallel H(pk))$
$c = Kyber.CPAPKE.Enc(pk, m, r)$
$K = KDF(\bar{K} \parallel H(c))$
return (c, K)

The encapsulation method generates a random key out of a hash of the hash of random selected bytes concatenated with a hash of the public key, and encrypts this new key with the CPAPKE encryption method. The final key, to be used on a symmetric algorithm is generated through a key derivation function with the random key and a hash of its encryption.

Algorithm 9. Kyber.CCAKEM.Dec(c, sk)

Input: Ciphertext c
Input: Secret key sk
Output: Shared key K
$(sk', pk, H(pk), z) = sk$
$m' = Kyber.CPAPKE.Dec(sk', c)$
$(\bar{K}', r') = G(m' \parallel H(pk))$
$c' = Kyber.CPAPKE.Enc(pk, m', r')$
if $c = c'$:
return $K = KDF(\bar{K}' \parallel H(c))$
else:
return $K = KDF(z \parallel H(c))$

For the last algorithm, the decapsulation, we can see some details of extreme importance for Kyber's final algorithm. Caused by the FO transform, there is a small chance of failure possible on the algorithm, this chance is estimated to be around the parameter δ for each security level. For this reason the decapsulation algorithm features not only a decryption, but also an encryption where the ciphertext received and calculated are compared to know if a failed encapsulation occurred. In case of a success, a key that is gonna be generated through a key derivation function is gonna be shared between both individuals.

3.5. Underlying mathematical problem

What truly makes Kyber to be considered a secure cryptographic scheme is the underlying lattice mathematical problem. Following ideas of linear algebra, we can

consider the main equation on Kyber key generation, A*s + e = t, as a lattice of base A, with a selected point defined by s, summed with an error term to reach a point out of the lattice t.

The security behind the algorithm is based that, given the point t and the base A (public key), it is computationally hard to find the closest point that belongs to that lattice, the point s (secret key).

3.6. Kyber Variants

There are 2 different variants of Kyber, that define different symmetric algorithms for the pseudo-random functions (PRF), extendable output function (XOF), two hashing functions (H and G) and a key-derivation function (KDF)

The "normal" variant utilizes primitive functions standardized by the FIPS-202 [3], so that XOF instantiates SHAKE-128, H instantiates SHA3-256, G instantiates SHA3-512, PRF(s, b) instantiates SHAKE-256(s || b) and KDF instantiates SHAKE-256.

The "90s" variant utilizes older standard functions so that XOF instantiates AES-256 in CTR mode, H instantiates SHA-256, G instantiates SHA-512, PRF(s, b) instantiates AES-256 CTR mode so that s is used as the key and b is used as a zero-padded nonce and KDF instantiates SHAKE-256.

The "normal" variant of Kyber offers newer, more efficient symmetric primitives to be used, while the "90s" variant offers primitives with more availability for hardware support. This is shown when we discuss about Kyber's performance where the "normal" variant scores better for a non-hardware accelerated implementation while the "90s" variant can make a better use of modern processor instructions for symmetric cryptographic primitives.

4. Kyber's Performance

4.1. Memory Usage

The memory usage on Kyber is as follows:

Table 2. Kyber Memory Usage (bytes)

	Kyber512	Kyber765	Kyber1024
Public Key	800	1184	1568
Secret Key	1632	2400	3168
Ciphertext	768	1088	1568

We can see that the secret key is the most memory expensive of the three, but, in the case of a memory sensitive implementation such as small hardware, the secret key can be kept as the seeds used to generate it instead, meaning a higher computation will be necessary but the memory consumption of the secret key

reduced to only 32 bytes. For the public key we can compare it to modern day public cryptography according to table 3.

Table 3. Public key memory usage of RSA and ECC (bytes)

Similar Security to:	RSA	ECC
Kyber512	384	32
Kyber768	960	48
Kyber1024	1920	64

When compared to the RSA, Kyber's public keys can be considered similar, even being smaller at the highest security level, but ECC extremely small key sizes makes it very hard for Kyber to be able to compete against it.

4.2. Execution Time

For the execution time we can obtain a great comparison between Kyber and current standard public cryptographic methods through [4], as we get the following table.

Table 4. Execution Time of Kyber and ECDH (ms)

	KeyGen	Encapsulation	Decapsulation
ECDH-secp256k1	42,79	85,57	42,79
ECDH-secp256r1	60,56	121,15	60,57
Kyber512	5,37	6,81	6,49
Kyber768	10,19	11,98	11,46
Kyber1024	16,41	18,58	17,85

This data shows how much faster Kyber is, being even faster than current elliptic curve cryptographic, which is considered to be very fast.

Next, we can compare the difference between Kyber's variants and between reference code and AVX2 code, obtained from [1].

Table 5. CPU Cycles of Kyber's variants on reference and AVX2 code

	KeyGen	Encaps	Decaps
Kyber512 - Reference	122684	154524	187960
Kyber512 90s - Reference	213156	249084	277612
Kyber512 - AVX2	33856	45200	34572
Kyber512 90s - AVX2	21880	28592	20980
Kyber768 - Reference	199408	235260	274900
Kyber768 90s - Reference	389760	432764	473984

Kyber768 - AVX2	52732	67624	53156
Kyber768 90s - AVX2	30460	40140	30108
Kyber1024 - Reference	307148	346648	396584
Kyber1024 90s - Reference	636380	672644	724144
Kyber1024 - AVX2	73544	97324	79128
Kyber1024 90s - AVX2	43212	56556	44328

Here we can see the differences between both variants and implementations that take advantage of AVX2 instructions. There are 2 main things to be noted from this data, the great increase of performance when using AVX2 instructions on both variants, and how the 90s and the normal variants change from most optimal depending if we are using AVX2 optimized code or reference code, as expected.

5. Parameter Selections

The parameter n defines the size of our polynomials, it is chosen as 256 to guarantee an encapsulation of keys with 256 bits of entropy. The parameter k fixes the dimension of the underlying lattice problem, being the main mechanism to scale security between Kyber512, Kyber768 and Kyber1024. The parameter q must be a small prime, such that q − 1 must be divisible by n as this is required to enable the fast NTT-based multiplication, given the choices available (257, 769, 3329, ...) it has been chosen as 3329 since the smaller primes would not be able to achieve a small enough failure probability. The parameters $\eta_{\{1,\ 2\}}$ defines the noise of s, e and r through the CBD function, being chosen a number so that the error coefficients are enough to make the problem hard to solve without compromising the possibility of decrypting a message. The parameters $d_{\{u,\ v\}}$ are chosen to balance security levels, ciphertext size and failure probabilty.

References

[1] Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck J. M., Schwabe, P., Seiler, G. and Stehle, D. (2020) CRYSTALS-Kyber Algorithm Specifications And Supporting Documentation (version 3.0)

[2] Fujisaki, E. and Okamoto, T. (1999) “Secure Integration of asymmetric and symmetric encryption schemes”, In: Advances in Cryptology – CRYPTO ‘99, p. 537-554

[3] National Institute of Standards and Technology (2015) FIPS PUB 202 – SHA-3 standard: Permutation-based hash and extendable-output functions, http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf

[4] Saarinen, M.-J., (2020) Mobile Energy Requirements of the Upcoming NISTPost-Quantum Cryptography Standards, https://arxiv.org/abs/1912.00916