0.1 Kyber

In the following section, we will briefly discuss the overall structure of the post-quantum Key-Encapsulation Mechanism (KEM) Kyber, which has recently been chosen by NIST as a part of the next Post-Quantum Cryptography standard. For a better understanding, one should take a look at the algorithms in the original Kyber specification [?].

0.1.1 Instantiating Kyber

Kyber is instantiated using the following parameters

- **d**: Maximal degree of the polynomials. Note that in the Kyber specification, this would be denoted n, but we have changed this to avoid conflicting notation.
- **k**: Number of polynomials in the vectors of the underlying MLWE problem.
- **q**: Modulus for all coefficients and numbers.
- $\boldsymbol{\eta_1}, \boldsymbol{\eta_2}$: Says something about how large the "small coefficients," mostly used by the error vector, can be.
- $\mathbf{d_u}, \mathbf{d_v}$: Controls the amount of compression on the ciphertext ($\mathbf{u}$, v).
- $\boldsymbol{\delta}$: Denotes the probability that the decryption fails.

The following three parameter sets are used in the standard Kyber specification. Note that while they provide a different amount of security, they are easily extendable with only minor changes between the parameter sets, as can further be seen in the TPKE parameter sets shown later in Table **??**.

Kyber parameters							
name	d	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}

Figure 1: Parameters in the original Kyber scheme

0.1.2 Symmetric primitives

For the sake of completeness, Kyber introduces two different instantiations for the symmetric primitives, as described below. One uses a more modern approach, for which new hardware is needed, and another where functions have been specifically handpicked to currently available hardware.

Modern

A modern instantiation of the functions is as follows. To instantiate the functions `PRF`, `XOF`, `H`, `G` and `KDF`, we use hash functions defined in the FIPS-202 standard [?]. All these functions can be obtained by a call to the Keccak function with appropriate arguments.

- `XOF` is instantiated with SHAKE-128
- `H` is instantiated with SHA3-256
- `G` is instantiated with SHA3-512
- `PRF(s, b)` is instantiated with SHAKE-256(s || b)
- `KDF` is instantiated with SHAKE-256

"90s" variant

The idea behind the choices in this "90s" variant is to choose already standardized functions by NIST, which have existing hardware implementations. These usually boil down to SHA-256 and AES.

- `XOF`(p, i, j) is instantiated with AES-256 in CTR mode where p is used as the key and $i||j$ padded with zeros to become a 12-byte nonce. The counter should start at 0.
- `H` is instantiated with SHA-256
- `G` is instantiated with SHA-512
- `PRF`(s, b) is instantiated with AES-256 in CTR mode where s is used as the key and b is padded to a 12-byte nonce. The counter should start at 0.
- `KDF` is instantiated with SHA-256

Note also that all the symmetric primitives for the modern variant could have been instantiated using only a single hash function, but that the authors of Kyber made an active choice to choose different functions from the FIPS-202 standard [?] to avoid having to prefix a character with the input to get domain separation.

0.1.3 Compression and decompression

Kyber defines two functions `Compress` and `Decompress` for compressing and decompressing values, effectively removing some of the low-order bits, which does not have much effect on the failure probability of the decryption, making ciphertexts smaller. For this, we use $l < \lceil \log_2 q \rceil$. `Compress` and `Decompress` are defined as follows

$$\texttt{Compress}_q(x, l) = \lceil (2^l/q) \cdot x \rfloor \bmod 2^l$$
$$\texttt{Decompress}_q(x, l) = \lceil (q/2^l) \cdot x \rfloor \bmod 2^l$$

This compression is lossy, and if we define an element x' by

$$x' = \texttt{Decompress}_q(\texttt{Compress}_q(x, l), l)$$

then we say that x and x' are close iff

$$|x' - x \bmod q| \leq B_q = \lceil q/2^{l+1} \rfloor$$

While the compression does compress the ciphertexts, making them smaller, it also handles the LWE error correction when encrypting and decrypting. In the encryption function, described in algorithm 5, the decompress function creates error tolerance gaps by sending a message bit 0 to the number 0 and 1 to the number $q/2$, as we know is necessary from usual LWE. Compress will reverse this in the decryption function based on whether $v - \boldsymbol{s}^T \cdot \boldsymbol{u}$, taken from algorithm 6, is closer to $\lceil q/2 \rceil$ or 0, correctly decoding to the corresponding bit.

0.1.4 Number Theoretic Transformation

The Number Theoretic Transformation (NTT) can be applied to polynomials in R_q to make polynomial multiplication faster. Initially, polynomial multiplication in the ring can be done in $\mathcal{O}(n^2)$, as each coefficient of either polynomial have to be multiplied with each coefficient in the other polynomial.

$$c = a \cdot b = (a_0 + a_1 x + \dots) \cdot (b_0 + b_1 x + \dots) = a_0 \cdot b_0 + a_0 \cdot b_1 x + \dots$$

By utilizing the NTT transformation, we can instead use some cleverly constructed algorithms to perform polynomial multiplication with complexity $\mathcal{O}(n \cdot log(n))$.

The overall idea is to split polynomials into many polynomials of smaller degrees, which by the Chinese Remainder Theorem, can then be assembled back together, to get the original polynomial. By reducing polynomials into smaller degrees, they will have fewer coefficients and can thus be multiplied faster.

In Kyber, we can represent the modulus polynomial as

$$x^{256} + 1 = \prod_{i=0}^{127} x^2 + \zeta^{2i+1} = \prod_{i=0}^{127} x^2 + \zeta^{2 \cdot br_7(i)+1}$$

where $br_7(i)$ is unsigned 7-bit reverse for any integer $i \in \mathbb{N}$ and the ζ values are the primitive 256'th roots of unity. The NTT of any polynomial $f \in R_q$ can then be represented by 128 polynomials as

$$(f \bmod x^2 + \zeta^{2 \cdot br_7(0)+1}, f \bmod x^2 + \zeta^{2 \cdot br_7(1)+1}, \ldots, f \bmod x^2 + \zeta^{2 \cdot br_7(127)+1})$$

Any polynomial $f \in R_q$ can then be defined by the following

$$\texttt{NTT}(f) = f_0 + f_1 x + f_2 x^2 + f_{255} x^{255}$$

where it holds that

$$f_{2i} = \sum_{j=0}^{127} f_{2j} \zeta^{(2 \cdot br_7(i)+1) \cdot j}$$
$$f_{2i+1} = \sum_{j=0}^{127} f_{2j+1} \zeta^{(2 \cdot br_7(i)+1) \cdot j}$$

Note here that we have defined a new polynomial with the same degree as the original polynomial but that this NTT version of the polynomial should not be seen as a polynomial in R_q. It's simply a construction to reuse the already present polynomial data structure, which is most relevant when implementing Kyber.

There exist no 512'th primitive roots for the specific choices of parameters in the Kyber scheme, and hence we cannot split the input polynomial into 256 polynomials. For the specific choices of Kyber, we have the root of unity $\zeta = 17$ as $x^{256} \equiv 1 \bmod 3329$ for $x = 17$.

By using this, we can now do multiplications on polynomials $f, g \in R_q$ by computing

$$\texttt{NTT}^{-1}(\texttt{NTT}(f) \cdot \texttt{NTT}(g))$$

which means that each coefficient can be calculated as

$$h_{2i} + h_{2i+1} x = (f_{2i} + f_{2i+1} x)(g_{2i} + g_{2i+1}) \bmod x^2 + \zeta^{2 \cdot br_7(i)+1}$$

This technique can easily be generalized to polynomial vectors as well by converting each polynomial in the vector to its corresponding NTT form, then applying the above transformation to each polynomial.

0.1.5 Montgomery domain

Let n be a positive integer and R, T be integers for which it holds that $R > n$, $gcd(n, R) = 1$ and $0 \leq T < n \cdot R$. Then the montgomery reduction of T modulo n with respect to R is defined by

$$T \cdot R^{-1} \bmod n$$

Note that in many implementations, it makes sense to choose the factor R as a multiple of 2 to make use of efficient bit shifting.

If we wish to calculate

$$c \equiv a \cdot b \bmod n$$

then instead of working on a and b directly, we can then draw them into the Montgomery domain by applying

$$\overline{a} = a \cdot R^{-1} \bmod n$$
$$\overline{b} = b \cdot R^{-1} \bmod n$$

Which uses a small overhead to convert a and b into $\overline{a}$ and $\overline{b}$, hoping that the speedup will make up for this at a later stage. To do addition and subtraction, we can do as usual

$$\begin{aligned}\overline{c} &= \overline{a} + \overline{b} \bmod n \\ &= (a \cdot R^{-1}) + (b \cdot R^{-1}) \bmod n \\ &= (a + b) \cdot R^{-1} \bmod n\end{aligned}$$

and similarly, we can do subtraction the trivial way.

Montgomery reduction

Whenever we have two numbers on the Montgomery form, we will use the following Montgomery reduction algorithm.

```
montgomery_reduce(a)
1 :  $u \leftarrow a \cdot q^{-1}$
2 :  $t \leftarrow u \cdot q$
3 :  $t \leftarrow a - t$
4 :  Return $t >> 16 \bmod q$
```

Figure 2: Montgomery reduction

Barrett reduction

When doing multiple reductions, the Barret reduction algorithm seen in Figure 3 is advantageous to use in Kyber. The algorithm works best when $a < n^2$ and is in the sense of Kyber used only when recreating a polynomial from its NTT form.

```
barrett_reduction(a)
1 :  $v \leftarrow ((1 >> 26) + q/2)/q$
2 :  $t \leftarrow (v \cdot a) + (1 << 25) >> 26$
3 :  $t \leftarrow t \cdot q$
4 :  Return $a - t$
```

Figure 3: Barrett reduction

0.1.6 Algorithms

In the following section, we will present the different algorithms included in the Kyber specification. For the full algorithms, see appendix **??**.

1. Parse: Kyber samples polynomials in R_q deterministically by using the `Parse` algorithm. `Parse` takes as input a byte stream $\mathcal{B} = b_1, b_2, \ldots, b_8$ and outputs an element $\hat{a} = \hat{a}_0 + \hat{a}_1 x + \hat{a}_2 x^2 + \cdots + \hat{a}_{255} x^{255}$ in R_q in its NTT form.

The intuition behind `Parse` is that it should output something statistically close to uniformly random in the ring R_q, given a byte stream that is statistically uniformly random. Note that by making `Parse` deterministic, it becomes possible to recreate the exact sequence of polynomials by giving the same byte input. This is a property that will be used when generating the matrix of polynomials $\boldsymbol{A}$.

2. CBD: Noise in Kyber is sampled from a centered binomial distribution (`CBD`) parametrized by η, which determines the size of the coefficients in the "small" polynomials sampled. When sampling a vector $f \in R_q$ from `CBD`, we mean that each individual coefficient will get sampled from `CBD`. Note that this function is very carefully chosen, such that we do not end up with too much noise in the underlying MLWE scheme and, as a consequence, is very specific to the chosen modulo q.

3. Decode / Encode: As the input and output types of Kyber are in bytes and byte streams, we need some kind of mechanism for converting byte streams to polynomials and vice versa. Note that in the original Kyber specification, there is no definition of how to encode polynomials, only how to decode, but we will describe both.

The `Encode` function of Kyber establishes a way of converting any polynomial into a series of $256 \cdot 12 = 3072$ bits or equivalently 384 bytes. It does so by encoding each coefficient of the polynomial in 12 bits and appending all 3072 bits together to form one large byte stream with each coefficient appended. A visual representation of this can be seen in Figure 4.

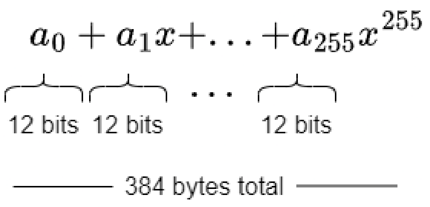

Figure 4: Visualisation of the Encode algorithm in Kyber

Trivially the `Decode` function takes this byte stream and converts it back into the individual coefficients by converting the chunks of 12 bits into each coefficient.

It's important to note that 12 bits are very consistently chosen with the standard Kyber modulus 3329 as $2^{11} < 3329 < 2^{12}$. To not lose any information, the number of bits can be no lower than 12 bits. For a larger modulus, one could just increase the number of bits to use for encoding/decoding in a trivial manner.

4. IND-CPA.KeyGen: To generate the keys, Kyber starts by using the `Parse` function to generate a matrix $A \in R_q^{k \times k}$ in which there are polynomials on the NTT form. The matrix A will be a square matrix of length k.

Afterward, the secret $\boldsymbol{s} \in R_q^k$ and error vector $\boldsymbol{e} \in R_q^k$ are chosen at random using the `CBD` function with parameter η_1. This results in the two vectors of polynomials with coefficients in the interval $[-\eta_1, \eta_1]$. So, again, the coefficients are small to ensure that there is not too much noise to be able to decrypt.

Using these, the public key and secret key are defined by the following

$$pk = (\mathbf{A}, \mathbf{t} = \mathbf{A} \cdot \mathbf{s} + \mathbf{e})$$
$$sk = \mathbf{s}$$

The public key and the secret key can be converted into byte streams by using the `Encode` functionality.

Note that Kyber utilizes different symmetric primitives to regenerate seemingly random values from the randomly chosen coins. Inserting the same coins will yield the same output. This is especially used when calculating the Matrix A as A does not need to be sent, only the randomness used to generate A, which is much smaller.

5. IND-CPA.Encrypt: The Encrypt algorithm encrypts a binary stream $m \in \mathcal{B}^{32}$ by using the public key and some random coins. These coins are used such that IND-CPA encryption can be used deterministically.

To encrypt a byte stream $m \in \mathcal{B}^{32}$ the matrix $A \in R_q^{k \times k}$ is sampled using the public key, from which the randomness initially used to generate the matrix A lies.

Randomness $\boldsymbol{r} \in R_q^k$ is sampled from `CBD` with parameter η_1. While the error vectors $\boldsymbol{e}_1 \in R_q^k$ and $e_2 \in R_q$ are sampled using `CBD` with parameter η_2. Using all these values, one can now encrypt by calculating

$$\boldsymbol{u} = \mathbf{A}^T \mathbf{r} + \mathbf{e}_1$$
$$v = \mathbf{r}^T \mathbf{t} + e_2 + \lfloor q/2 \rceil \cdot m$$

then compressing and using the `Encode` to convert $(\boldsymbol{u}, v)$ into a byte stream, appending v on $\boldsymbol{u}$.

6. IND-CPA.Decrypt: Decrypting any ciphertext $(\boldsymbol{u}, v)$ into the message $m \in \mathcal{B}^{32}$, encrypted by the `Encrypt` functionality is done by using the secret keys $\boldsymbol{s}$ and calculating

$$c' = v - \mathbf{u}^T \mathbf{s}$$

then compressing and encoding the result to get the correct byte stream. This has the effect of setting the bits of the output message to be the coefficients of the binary polynomial computed by $\lfloor c' \cdot 2/q \rceil$.

7. KeyGen: Using a slightly tweaked Fujisaki-Okamoto transformation, Kyber transforms its IND-CPA scheme, as defined above, into one resistant against attacks in which the ciphertexts can be chosen; more specifically, we get an IND-CCA2 secure scheme.

The algorithm samples some random bytes $z \in \mathcal{B}^{32}$, then uses the `IND-CPA.KeyGen` to receive public key `pk` and secret key `sk`. From this, the secret key is defined as

$$sk' = (sk || pk || H(pk) || z)$$

Where `H` is a hash function defined by the symmetric primitives.

8. Encapsulate: `Encapsulate` is used by a party to generate a key K and a ciphertext c, to send to the other party.

First, 32 random bytes $m \in \mathcal{B}^{32}$ are sampled, then m is updated to be $\mathtt{H}(m)$. Then the key $\overline{K}$ and randomness r are derived using `G` by computing $(\overline{K}, r) := G(m || H(pk))$. Using the randomness r, we utilize `IND-CPA.Encrypt` to encrypt m under the public key pk, which yields the ciphertext c. The key is then computed as $K = \mathtt{KDF}(\overline{K} || H(c))$. Finally, the output is the ciphertext and key (c, K), where the ciphertext can be sent to the recipient.

9. Decapsulate: When receiving the ciphertext c from `Encapsulate`, the recipient wishes to calculate the same key K as the sender has gotten from `Encapsulate`.

To do this, apply the `IND-CPA.Decrypt` function on c using the secret key `sk` to obtain m', which concatenated with `H` applied on the public key, can be used as input to function `G`, returning $(K', r') := G(m'||H(pk))$. Afterward, a check for the correct ciphertext c' is done by re-encrypting the found message m' using the public key pk and the randomness r' found earlier. If $c = c'$ then `Decapsulate` will return $K := \texttt{KDF}(K'||H(c))$ otherwise $K := \texttt{KDF}(z||H(c))$.