The Cryptography Caffè

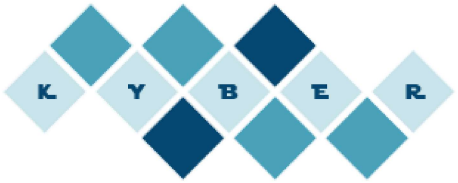
Adventures in PQC: Exploring Kyber in Python - Part I

by Steven Yue, James Howe. Posted on Nov 30, 2022

"A picasso drawing of Alice delivering a shared secret to Bob using CRYSTALS-Kyber cryptosystem" by DALL-E.

0. Preface

CRYSTALS-Kyber, or simply Kyber, is a key encapsulation mechanism (KEM) selected by NIST as its first KEM standard for resistance against a powerful, quantum adversary, an area generally referred to as Post-Quantum Cryptography (PQC). In this series of blog posts, we will look into how Kyber works under the hood and use Python to build Kyber from the ground up.



CRYSTALS-Kyber Logo

Before we begin, let's take a look at the formal description of Kyber. Kyber is a lattice-based cryptosystem; meaning that the hardness problem securing Kyber is a lattice problem, specifically the hardness of solving the Learning With Errors (LWE) problem in module lattices (MLWE) [1]. Kyber is a public-key (asymmetric) cryptosystem. One of its primary design purposes is to establish a shared secret between two communicating parties (which then continue using symmetric cryptography) in a secure way without an attacker being able to decrypt it (and thus future communications between the parties).

Kyber offers (i) a CPA-secure public-key encryption (PKE) scheme, which at its roots is tied to seminal works by Regev [2] and the Lyubashevsky, Peikert, and Regev [3] (LPR) encryption scheme, (ii) as well as an IND-CCA2-secure KEM by using the PKE scheme combined with the Fujisaki-Okamoto (FO) transform [4].

Kyber offers three sets of parameters, named Kyber512, Kyber768, and Kyber1024, with the aim of achieving 128, 192, and 256 bit security levels, respectively. A noteworthy quality of Kyber's design is how it scales these parameters; unlike Ring-LWE encryption schemes and the previous candidate NewHope [5], which change the lattice dimension (n), modulus (q), and variance of the distribution of the noise to increase security levels, Kyber instead fixes these values and achieves higher security by increasing a scaling parameter, k, to in-turn attain a higher lattice dimension. This means operations within Kyber, such as multiplication, can all be done using the same fundamental operations, using the same parameters, except that more repetitions of these same operations are required to achieve a higher level of security.

Above is a quick summary of the properties and functionalities of Kyber. Although the description might look intimidating, we will assure you that it's not through a few toy examples and gradually ramping up the complexity.

In this blog post, we will go through some of the fundamentals of Module LWE, and build up KYBER from scratch step-by-step.

1. (Re)introduction to Polynomials

Kyber works in the space of polynomials. Every arithmetic operation is between polynomials, vectors of polynomials, or matrices of polynomials - just like how we deal with numbers.

Therefore, to better understand how Kyber works, we need to figure out the polynomial math first.

Although it might sound scary, I will assure you it's not. In fact, the best way to understand polynomials in the context of computer science is that they are just **fancy lists of numbers** with some special ways to do arithmetic.

Representation

Normally, a polynomial would look something like:

$$P(X) = 5X^3 + 2X^2 + 1$$

or, more generically, in the form of:

$$P(X) = \sum_{i=0}^{n} p_i X^i$$

where p_is are coefficients and n is the max degree of the polynomial.

Therefore, naturally, we can represent a polynomial with only its $n+1$ coefficients:

$$[p_0, p_1, \ldots, p_n]$$

For example, the $P(X)$ above can be represented as $[1, 0, 2, 5]$.

Fields and Rings

Before we go into the arithmetic, we will take a look at one more requirement to make sure our computation is always bounded - we want to constrain our system with two more algebraic structures.

First, numbers in Kyber are defined in the finite field $GF(q) = \mathbb{Z}/q\mathbb{Z}$. Here $\mathbb{Z}$ is the set of all integers, q is a prime, and GF stands for Galois Field. However, in simple words, the field only contains integers, and it just defines a plain modulus q, and every operation to a number or polynomial coefficient in Kyber will take modulo q in the end.

Second, polynomials in Kyber are defined in another special structure called a polynomial ring $R = GF(q)[X]/(X^n + 1)$. The notation might look intimidating, but what it really means is that every polynomial operation will take modulo $X^n + 1$ in the end.

In order to compute the modulo operation with a polynomial, the traditional approach is to perform a long division (like the one we learn in school) of the target polynomial by $X^n + 1$ and save the remainder as the final result. However, since our modulus is $X^n + 1$, it means that $X^n = -1$ as the ring wraps around. Therefore, we can simply replace every X^n with a -1 in the target polynomial, until all the terms have smaller order than n.

Polynomial Operations

After understanding the basic arithmetic setup of Kyber, we can start by implementing some of the most simple polynomial operations, which will become handy for later use.

- **Addition**: Adding two polynomials is quite straightforward, we simply add each coefficient together modulo q. Namely:

$$P(X) + Q(X) = \sum_{i=0}^{n} (p_0 + q_0 \bmod q) X^i$$

- **Subtraction**: Subtracting one polynomial $Q(X)$ from another one $P(X)$ is equivalent to inverting polynomial $Q(X)$ by changing the signs on every coefficient q_i. And then take modulo q to obtain a non-negative number, and perform **Addition** above.

$$P(X) - Q(X) = \sum_{i=0}^{n} (p_0 - q_0 \bmod q) X^i$$

- **Multiplication**: Multiplication of $P(X), Q(X)$ works in the same way as schoolbook multiplication where we multiply every component $p_i X^i$ in $P(X)$ with $Q(X)$ and sum them up in the end. For the final result, not only do we need to take modulo q for all the coefficients, we also need to take modulo $F(X) = X^n + 1$ for the final polynomial such that we obtain a result with degree $n - 1$. The modulo operation can be done simply via a long division of $P(X) \cdot Q(X)$ by $F(X)$ and take the remainder.

```
py
import numpy as np
from numpy.polynomial.polynomial import Polynomial

import random
```

```
py
def add_poly(a, b, q):
  # adds two polynomials modulo q
  result = [0] * max(len(a), len(b))
  for i in range(max(len(a), len(b))):
    if i < len(a):
      result[i] += a[i]
    if i < len(b):
      result[i] += b[i]
    result[i] %= q
  return result


def inv_poly(a, q):
  return list(map(lambda x: -x % q, a))


def sub_poly(a, b, q):
  return add_poly(a, inv_poly(b, q), q)


def mul_poly_simple(a, b, f, q):
  tmp = [0] * (len(a) * 2 - 1) # the product of two degree n polynomial cannot exceed 2n

  # schoolbook multiplication
  for i in range(len(a)):
    # perform a_i * b
    for j in range(len(b)):
      tmp[i + j] += a[i] * b[j]

  # take polynomial modulo f
  # since Kyber's poly modulus is always x^n + 1,
  # we can efficiently compute the remainder
  degree_f = len(f) - 1
  for i in range(degree_f, len(tmp)):
    tmp[i - degree_f] -= tmp[i]
    tmp[i] = 0

  # take coefficients modulo q
  tmp = list(map(lambda x: x % q, tmp))
  return tmp[:degree_f]
```

Testing Polynomial Operations

The code above implements the polynomial operations such as addition, subtraction, and multiplication in a finite field $GF(q)$ and a polynomial ring $\mathbb{R}$.

Addition and subtraction are fairly straightforward. To validate the correctness of the multiplication implementation, we randomly sample some polynomials, compute their product using our implementation, and compare that against the reference implementation result from NumPy's Polynomial class.

Note that in production implementations, it would be nicer to compare the results against a more formal math library, such as SageMath or NTL, to make sure all the field/ring arithmetics are done correctly.

```
py
np.random.seed(0xdeadbeef)

def sign_extend(poly, degree):
  if len(poly) >= degree:
    return poly

  return [0] * (degree - len(poly))

def test_mul_poly(N, f, q):
  degree_f = len(f) - 1

  for i in range(N):
    a = (np.random.random(degree_f) * q).astype(int)
    b = (np.random.random(degree_f) * q).astype(int)

    a_mul_b = mul_poly_simple(a, b, f, q)

    # NumPy reference poly multiplication
    # note that we need to convert the coefficients to int and extend the list to match the fixed size of our impl
    a_mul_b_ref = list(map(lambda x: int(x) % q, ((Polynomial(a) * Polynomial(b)) % Polynomial(f)).coef))
    a_mul_b_ref = sign_extend(a_mul_b_ref, degree_f)

    assert(a_mul_b == a_mul_b_ref)

test_mul_poly(100, [1, 0, 0, 0, 1], 17)
```

Vectorize Polynomials

Once we've defined the simple polynomial operations, we can naturally create vectors and matrices made of polynomials, and define vector/matrix addition and multiplication.

- **Vector Addition**: This is simply component-wise polynomial addition.
- **Vector Multiplication (Inner Product)**: This is simply a component-wise polynomial multiplication and summing up the results using polynomial addition.
- **Matrix-Vector Multiplication**: This works exactly like the plain Matrix-Vector Multiplication, where we perform Vector Multiplication for every row of the left-hand term with every column of the right-hand term.

```
py
def add_vec(v0, v1, q):
  assert(len(v0) == len(v1)) # sizes need to be the same

  result = []

  for i in range(len(v0)):
    result.append(add_poly(v0[i], v1[i], q))

  return result


def mul_vec_simple(v0, v1, f, q):
  assert(len(v0) == len(v1)) # sizes need to be the same

  degree_f = len(f) - 1
  result = [0 for i in range(degree_f - 1)]
```

```
    # textbook vector inner product
    for i in range(len(v0)):
      result = add_poly(result, mul_poly_simple(v0[i], v1[i], f, q), q)

    return result


def mul_mat_vec_simple(m, a, f, q):
    result = []

    # textbook matrix-vector multiplication
    for i in range(len(m)):
      result.append(mul_vec_simple(m[i], a, f, q))

    return result


def transpose(m):
    result = [[None for i in range(len(m))] for j in range(len(m[0]))]

    for i in range(len(m)):
      for j in range(len(m[0])):
        result[j][i] = m[i][j]

    return result
```

Testing Vector Operations

Similar to the previous test section, we randomly sample matrices and vectors and compare our multiplication result against NumPy's reference result.

```
py
np.random.seed(0xdeadbeef)

def test_mul_vec(N, k, f, q):
    degree_f = len(f) - 1

    for i in range(N):
      m = (np.random.random([k, k, degree_f]) * q).astype(int)
      a = (np.random.random([k, degree_f]) * q).astype(int)

      m_mul_a = mul_mat_vec_simple(m, a, f, q)

      m_poly = list(map(lambda x: list(map(Polynomial, x)), m))
      a_poly = list(map(Polynomial, a))
      prod = np.dot(m_poly, a_poly)
      m_mul_a_ref = list(map(lambda x: list(map(lambda y: int(y) % q, sign_extend((x % Polynomial(f)).coef, degree_f))), prod))

      assert(m_mul_a == m_mul_a_ref)

test_mul_vec(100, 2, [1, 0, 0, 0, 1], 17)
```

2. Inner Kyber Public-Key Encryption

Now we have defined all the underlying primitives that we need, we can move forward to implement the underlying PKE primitive of Kyber!

Key Generation

First of all, let's talk about generating keys for Kyber.

In the previous section, we have seen basic matrix-vector operations with polynomials in a finite field and a polynomial ring. Kyber is exactly built on top of the same arithmetic foundation!

Assume we randomly sample a matrix A with dimension $k \times k$ made with polynomials with the same arithmetic operations that we defined above. We additionally sample two other vectors s, e consisting of k polynomials such that the polynomials in them have "small coefficients".

Now, if we multiply A and s together, and then add e to it, we obtain:

$$t = As + e$$

We then denote (A, t) as the public key and s as the secret key.

Quantum-Resistant Security

Before we dive into how to utilize the keys, we can briefly take a look at the structure of the public key (A, t). Since t is not simply a matrix-vector product but also offset with a random "noise" vector, figuring out the value of the secret key s by just looking at (A, t) is not trivial. (If there were no noise offset, one can use row reduction and Gaussian elimination tricks to solve for s.)

More specifically, solving for s from (A, t) can be reduced to solving an instance of the problem of Module Learning With Errors (M-LWE), which is believed to be hard even for quantum computers. Hence, this is where Kyber gains quantum-resistance - by linking its core underlying public-key scheme with a mathematically hard problem (M-LWE) that cannot be efficiently solved by a quantum computer.

Encryption

Once we have generated a key pair, we can move onto encryption, where one party uses the public key (A, t) to encrypt a message m.

To perform encryption, the encrypting party will first draw a blinding vector $r \in R^k$, a new error vector $e_1 \in R^k$, and a single error polynomial $e_2 \in R$. All the polynomials being sampled will have "small coefficients" in the same way as s, e in the Key Generation above.

Next, the encrypting party will use the binary representation of the message being encrypted, and encode each bit as a coefficient in a binary polynomial. (For example, if $m = 11 = (1011)_2$, $m_b = X^3 + X + 1$.)

Finally, the Kyber inner PKE encryption scheme is defined as the following two computations that assemble the elements above together:

$$Enc((A, t), m_b) \rightarrow (u, v)$$

$$u = A^T r + e_1$$

$$v = t^T r + e_2 + \lfloor \frac{q}{2} \rceil m_b$$

In the end, we obtain a vector u and a single polynomial v as the ciphertext of the Kyber inner PKE. Note that the $\lfloor \frac{q}{2} \rceil m_b$ part of v is to simply scale up the binary polynomial m_b such that it has amplitude closest to half of the underlying finite field size q.

The scaling part is a crucial step to ensure successful decryption because the message is a binary message (0,1), and the underlying field q is much larger (for example 17). In that case, 1 is encoded as 9, which is much further from 0 (which still encodes to 0). By doing so, even when the final value gets perturbed by a small noise, we can still see if it's closer to 0 or 9 to decode whether it was a 0 or 1 from the beginning. The scaling of the binary values essentially gives us more room for errors while still allowing decryption to succeed.

We can quickly implement the `encrypt` functionality by directly computing u and v.

```
py
def encrypt(A, t, m_b, f, q, r, e_1, e_2):
  half_q = int(q / 2 + 0.5)
  m = list(map(lambda x: x * half_q, m_b))

  u = add_vec(mul_mat_vec_simple(transpose(A), r, f, q), e_1, q)
  v = sub_poly(add_poly(mul_vec_simple(t, r, f, q), e_2, q), m, q)

  return u, v
```

Decryption

Upon receiving the ciphertext, we can now take a look at how to decrypt the ciphertext and recover the original message.

Kyber defines the inner PKE decryption as the following:

$$
\begin{aligned}
Dec(s, (u, v)) &\rightarrow m_n \\
m_n &= v - s^T u \\
&= t^T r + e_2 + \lfloor \frac{q}{2} \rceil m_b - s^T (A^T r + e_1) \\
&= (As + e)^T r + e_2 + \lfloor \frac{q}{2} \rceil m_b - s^T (A^T r + e_1) \\
&= e^T r + e_2 + \lfloor \frac{q}{2} \rceil m_b - s^T e_1
\end{aligned}
$$

We can see that the remaining noise terms $e^T r, e_2, s^T e_1$ are all relatively small because they are sampled with "small coefficients". And on the other hand, the $\lfloor \frac{q}{2} \rceil m_b$ is relatively large since it's being scaled to have amplitude half the size of the field q.

Therefore, to recover m_b from m_n, we simply perform a "rounding" operation and see whether each coefficient in m_n is closer to $\lfloor \frac{q}{2} \rceil m_b$ or 0. Finally, the comparison results yields a boolean vector that should match to the initial m_b.

Similarly, we implement `decrypt` by computing m_n and manually doing the round operation.

```py
def decrypt(s, u, v, f, q):
  m_n = sub_poly(v, mul_vec_simple(s, u, f, q), q)

  half_q = int(q / 2 + 0.5)
  def round(val, center, bound):
    dist_center = np.abs(center - val)
    dist_bound = min(val, bound - val)
    return center if dist_center < dist_bound else 0

  m_n = list(map(lambda x: round(x, half_q, q), m_n))
  m_b = list(map(lambda x: x // half_q, m_n))

  return m_b
```

Observations

Before moving on to testing and validating the PKE scheme, here are a few observations that we can make to help us better understand what the PKE is essentially doing under the hood.

We see that the actual message (scaled m_b) is being scrambled by $t^T r + e_2$ which acts kind of like a one-time-pad with some additional noise. Due to the special structure of $t = As + e$, the party that possess the secret key s can approximately "reconstruct" the $t^T r$ portion of the one-time-pad from the $A^T r$ portion of u. Thus, the party can effectively remove the majority of the one-time-pad from v plus/minus some additional noise incurred by the noise vectors. But later, the scaling of the message and the rounding operations take care of the noise and recovers the message.

Testing Kyber inner PKE

Now that we have implemented the main routines of the Kyber inner PKE, we can now systematically test its functionalities and see if it's actually working or not.

First, we follow the same setup as Baby Kyber and plug in the test vectors and parameters the original blog post has provided.

```py
# Baby Kyber params
q = 17 # plain modulus
f = [1, 0, 0, 0, 1] # poly modulus, x**4 + 1

s = [[0, 1, -1, -1], [0, -1, 0, -1]] # secret key, [-x**3-x**2+x, -x**3-x]
A = [[[11, 16, 16, 6], [3, 6, 4, 9]], [[1, 10, 3, 5], [15, 9, 1, 6]]] # public key
e = [[0, 0, 1, 0], [0, -1, 1, 0]] # noise
m_b = [1, 1, 0, 1] # message in binary

t = add_vec(mul_mat_vec_simple(A, s, f, q), e, q)

r = [[0, 0, 1, -1], [-1, 0, 1, 1]] # blinding vector for encrypt
e_1 = [[0, 1, 1, 0], [0, 0, 1, 0]] # noise vector for encrypt
e_2 = [0, 0, -1, -1] # noise poly for encrypt
```

```
u, v = encrypt(A, t, m_b, f, q, r, e_1, e_2)
m_b2 = decrypt(s, u, v, f, q)

assert(m_b == m_b2)
```

Next, we will naively parametrize and randomize all aspects of the PKE scheme, and see if it works.

```
py
np.random.seed(0xdeadbeef)

def test_enc_dec(N, k, f, q):
  degree_f = len(f) - 1

  A = (np.random.random([k, k, degree_f]) * q).astype(int)
  s = (np.random.random([k, degree_f]) * 3).astype(int) - 1
  e = (np.random.random([k, degree_f]) * 3).astype(int) - 1
  t = add_vec(mul_mat_vec_simple(A, s, f, q), e, q)

  failed = 0

  for i in range(N):
    m_b = (np.random.random(degree_f) * 2).astype(int)

    r = (np.random.random([k, degree_f]) * 3).astype(int) - 1
    e_1 = (np.random.random([k, degree_f]) * 3).astype(int) - 1
    e_2 = (np.random.random([degree_f]) * 3).astype(int) - 1

    u, v = encrypt(A, t, m_b, f, q, r, e_1, e_2)
    m_b2 = decrypt(s, u, v, f, q)

    if m_b.tolist() != m_b2:
      failed += 1

  print(f"[k={k}, f={f}, q={q}] Test result: {failed}/{N} failed decryption!")

test_enc_dec(100, 2, [1, 0, 0, 0, 1], 17)
test_enc_dec(100, 2, [1, 0, 0, 0, 1], 37)
test_enc_dec(100, 2, [1, 0, 0, 0, 1], 67)
```

```
[k=2, f=[1, 0, 0, 0, 1], q=17] Test result: 27/100 failed decryption!
[k=2, f=[1, 0, 0, 0, 1], q=37] Test result: 1/100 failed decryption!
[k=2, f=[1, 0, 0, 0, 1], q=67] Test result: 0/100 failed decryption!
```

We can see that using the same Baby Kyber parameters ($q = 17$, small coefficients are $-1, 0, 1$), we have a significant amount of failed decryptions! However, as we increase the underlying modulus, the decryption success rate increases.

This is due to a few major reasons:

1. First, our noise sampling approach is not right. As we are just drawing random numbers and rounding them to integers, the actual distribution won't be the same ones that Kyber specified. We will not worry about this part for now until we get to the real sampling part.
2. Next, our noise-to-modulus ratio is perhaps not right. In order for the underlying M-LWE problem instance to be secure and have the right amount of hardness (for example, 2^{128}), the noise ratio parameters have to be extra carefully selected, as any deviation or convenience in implementation could lead to catastrophic vulnerabilities.

That being said, we do have a working public-key encryption scheme based on M-LWE problem that gives us a high success rate! The remaining part is to make it secure.

3. Next Steps

Now we have a rough shape of the PKE scheme that's used in Kyber (commonly referred as `InnerPKE`), here are the remaining steps that we need to do to make it work efficiently and securely:

1. Scale up the parameters (k, plain modulus q, polynomial modulus f, etc) such that it achieves post-quantum security.

2. Once we work with large parameters, the polynomial operations need to happen more efficiently than our schoolbook approaches. Therefore, Number-Theoretic Transform (NTT) needs to be used in order to effectively compute polynomial multiplications.

3. Instead of randomly sampling big matrices, we use a random seed and do pseudo-random expansion of the seed to populate the large data structures and conserve size.

4. To further shorten the ciphertexts, we need to look for compression techniques to pack the polynomial vectors by removing unimportant information and a "lossless" decompression that recovers to the original form.

Lastly, once we have perfected the `InnerPKE`, this is not the end yet! In fact, Kyber's `InnerPKE` would only give us a CPA-secure public-key encryption scheme. In order to be useful in several real-life scenarios, we need to convert it into a CCA-secure scheme.

In our next blog post, we will talk more about improving our current naive `InnerPKE` to be more compliant to Kyber's actual definition by focusing on the 4 points mentioned above. And after that we can take a look at how to convert the CPA-secure `InnerPKE` scheme to a CCA-secure KEM (Key Encapsulation Mechanism) via the Fujisaki-Okamoto Transform.

References

[1], Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. Designs, Codes and Cryptography, 75(3):565–599, 2015. https://eprint.iacr.org/2012/090.

[2], Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. Journal of the ACM, 56(6):34, 2009. http://www.cims.nyu.edu/~regev/papers/qcrypto.pdf.

[3], Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, Advances in Cryptology – EUROCRYPT 2010, volume 6110 of LNCS, pages 1–23. Springer, 2010. http://www.iacr.org/archive/eurocrypt2010/66320288/66320288.pdf.

[4], Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Advances in Cryptology - CRYPTO '99, pages 537–554, 1999. https://link.springer.com/chapter/10.1007/3-540-48405-1_34.

[5], Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange – a new hope. In Proceedings of the 25th USENIX Security Symposium, pages 327–343. USENIX Association, 2016. http://cryptojedi.org/papers/#newhope.