Kyber - How does it work?

2021-09-14 10 min read

This blog post describes the post-quantum cryptography (PQC) system Kyber, which is a finalist in NIST's PQC competition. In the first section I'll introduce post-quantum crypto and give some background on Kyber. Then I'll describe the concrete system by looking at a minified version. The last section will cover remaining technical details and security claims.

Kyber and Post-Quantum Crypto

Already in the early 2000s cryptographers got increasingly worried about potential advances in quantum computing. Since Peter Shor published his famous Shor's Algorithm we know that a large enough quantum computer would break all widely used public key systems. This includes RSA, finite field and elliptic curve constructions.

As a consequence, in 2017 the National Institute of Standards and Technology called for new public key systems that can withstand quantum computers. Kyber is such a proposed post-quantum scheme. In 2021 NIST decided it is worthy of standardization.

Kyber is a post-quantum public-key encryption system. Its main use case is to establish keys of symmetric-key systems in higher-level protocols like TLS, Signal or OpenPGP. it is a post-quantum system because Kyber is specifically designed to be secure even in the presence of quantum computers.

Kyber comes in three security levels. The size vs. security tradeoffs are shown in the following table with RSA as a pre-quantum comparison.

Version	Security Level	Private Key Size	Public Key Size	Ciphertext Size
Kyber512	AES128	1632	800	768
Kyber768	AES192	2400	1184	1088
Kyber1024	AES256	3168	1568	1568
RSA3072	AES128	384	384	384
RSA15360	AES256	1920	1920	1920

While RSA keys are still smaller, Kyber key sizes are in the same magnitute. This is not a given, as some PQC systems have keys in the hundreds of kilobytes, or in case of Classic McEliece even in the megabyte range.

Approachable Cryptography

To give an intuition about Kyber we'll start with a down-scaled version of the system, I call *Baby Kyber*. *Baby Kyber* is equivalent to "regular" Kyber, except that the security parameters are smaller, making everything much more readable. Also, "regular" Kyber uses compression for ciphertexts, which we will omit here as it is not relevant for the underlying crypto system.

Kyber is a public key encryption system. Hence, it works with public keys for encryption and private keys for decryption. As with other public key systems, we need a modulus q. That is, everytime we calculate with numbers we take the result modulo q. For *Baby Kyber* we specify this modulus as $q = 17$. Now, unfortunately with Kyber we have to deal with polynomials (a lot). Since we will mutiply and add polynomials, we also need a modulus for those. Otherwise their degree would become too big for us to handle. The polynomial modulus we'll use is $f = x^4 + 1$. For us it is not important why f looks the way it does. Important is the fact that by taking a polynomial modulo f, we guarantee that its degree (highest exponent) will be smaller than 4. Below all calculations are implicitly done modulo q (on coefficients) and f (on polynomials).

Key Generation

The **private key** of a Kyber key pair consists of polynomials with small coefficients, i.e. it is a vector of "small" polynomials. In *Baby Kyber* each private key contains two polynomials. In our example we'll use the private key:

$$\mathbf{s} = (-x^3 - x^2 + x, -x^3 - x)$$

Generating this private key is straightforward. We essentially use random small coefficients.

A Kyber **public key** consists of two elements. A matrix of random polynomials $\mathbf{A}$ and a vector of polynomials $\mathbf{t}$. Generation of the matrix is fairly simple, we just generate random coefficients and take them modulo q. For our example we'll use:

$$\mathbf{A} = \begin{pmatrix} 6x^3 + 16x^2 + 16x + 11 & 9x^3 + 4x^2 + 6x + 3 \\ 5x^3 + 3x^2 + 10x + 1 & 6x^3 + x^2 + 9x + 15 \end{pmatrix}$$

To calculate $\mathbf{t}$ we need an additional error vector $\mathbf{e}$. This error vector also consists of polynomials with small coefficients, exactly like the private key. In our example we'll use the error vector:

$$\mathbf{e} = (x^2, x^2 - x)$$

Now we can calculate $\mathbf{t}$ by matrix multiplication and additon:

$$t = \mathbf{A}\mathbf{s} + \mathbf{e}$$

We now have a *Baby Kyber* key pair with:

- Private key: $\mathbf{s}$
- Public key: $(\mathbf{A}, \mathbf{t})$

The trick is that it is a hard problem to recover $\mathbf{s}$ from $(\mathbf{A}, \mathbf{t})$. In fact, recovering $\mathbf{s}$ would require an attacker to solve the module-learning-with-errors (MLWE) problem, on which this system is built. The MLWE problem is expected to be hard even for quantum computers, which is why it is used in PQC.

Encryption

As in every public key encryption system, we can encrypt a message using the public key. Decryption can only be done by parties in possession of the private key. The encryption procedure uses an error and a randomizer polynomial vector $\mathbf{e_1}$ and $\mathbf{r}$. These polynomial vectors are freshly generated for every encryption. Additionally, we need an error polynomial $\mathbf{e_2}$. The polynomials within $\mathbf{e_1}$, $\mathbf{e_2}$ and $\mathbf{r}$ are completely random and small, just like the ones in $\mathbf{s}$.

In our example we'll use:

$$
\begin{aligned}
r &= (-x^3 + x^2, x^3 + x^2 - 1) \\
e_1 &= (x^2 + x, x^2) \\
e_2 &= -x^3 - x^2
\end{aligned}
$$

Now, to encrypt a message, we have to turn it into a polynomial. We do so by using the message's binary representation. Every bit of the message is used as a coefficient. In our example, we want to encrypt the number 11. Eleven has a binary representation of 1011, $(11)_{10} = (1011)_2$. Our message encoded as binary polynomial therefore is:

$$m_b = 1x^3 + 0x^2 + 1x^1 + 1x^0 = x^3 + x + 1$$

Before encryption we have to *scale* this polynomial. We *upscale* m_b by multiplying it with $\lfloor \frac{q}{2} \rceil$, i.e. the integer closest to $\frac{q}{2}$. This is done because the polynomial's coefficients need to be large. In the decryption part we'll see why this is necessary. In our example with $q = 17$, $\lfloor \frac{q}{2} \rceil = 9$. Our final ready-to-be-encrypted message therefore is:

$$m = \lfloor \frac{q}{2} \rceil \cdot m_b = 9 \cdot m_b = 9x^3 + 9x + 9$$

We encrypt m using the public key $(\mathbf{A}, \mathbf{t})$. The encryption procedure calculates two values $(\mathbf{u}, v)$.

In our example, those values are:

$$\mathbf{u} = (11x^3 + 11x^2 + 10x + 3, 4x^3 + 4x^2 + 13x + 11)$$
$$v = 7x^3 + 6x^2 + 8x + 15$$

Kyber ciphtertexts consist of those two values: ($\mathbf{u}$, v). A polynomial vector $\mathbf{u}$ and the polynomial v.

Decryption

Given the private key $\mathbf{s}$ and a ciphertext ($\mathbf{u}$, v), the decryption is straightforward. First, we compute a *noisy* result m_n.

$$m_n = v - \mathbf{s}^T\mathbf{u}$$

This result is *noisy*, because the computation actually does not yield the original message m. By looking at the equations we can see that:

$$m_n = \mathbf{e}^T\mathbf{r} + e_2 + m + \mathbf{s}^T\mathbf{e_1}$$

Now it becomes apparent why we needed to scale m by making its coefficients large. If you recall, all other terms in the equation were chosen to be small. So the coefficients of m_n are either close to $\lfloor \frac{q}{2} \rceil = 9$ implying that the original binary coefficient of m_b was a 1 or close to 0 implying the original binary coefficient was 0.

In our example we have $m_n = 7x^3 + 14x^2 + 7x + 5$. We can recover the original scaled message m by going through the coefficients of m_n and check if they are closer to $\lfloor \frac{q}{2} \rceil = 9$ or 0 (or equivalently q).

So, let's do that for all coefficients:

- 7, closer to 9 than 0 or p, round to 9
- 14, closer to p than 9, round to 0
- 7, closer to 9 than 0 or p, round to 9
- 5, closer to 9 than 0 or p, round to 9

Our rounded polynomial is $9x^3 + 0x^2 + 9x + 9$, which is the *scaled* polynomial that we encrypted! We can now simply recover the the original binary polynomial m_b by scaling down with factor $\frac{1}{9}$:

$$m_b = \frac{1}{9}(9x^3 + 0x^2 + 9x + 9) = (1x^3 + 0x^2 + 1x + 1)$$

Approachable Cryptography

Scaling Baby Kyber

The Kyber public key encryption systems works just like *Baby Kyber*, just with bigger parameters and compression. In fact, a Kyber parameter set is defined by the variables:

- n: maximum degree of the used polynomials
- k: number of polynomials per vector
- q, modulus for numbers
- η_1, η_2, control how big coefficients of "small" polynomials can be
- d_u, d_v, control how much $(\mathbf{u}, v)$ get compressed
- δ, shows the probability of a decryption yielding a wrong result

The parameter sets running for standrdization use the following variables:

Name	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}

As you can see, the main difference is in the vector size k. This is great from an implementer's perspetive, as all parameter sets can virtually use the same code. Highly optimized code (e.g. for working with the modulus q) can therefore be reused for all parameter sets!

Kyber KEM

Kyber is usually refered to as a KEM, not a public key encryption (PKE) system. The reason for this is somewhat technical. Let's just say Kyber uses a technique to turn the IND-CPA secure PKE into an IND-CCA2 secure KEM. For a developer, the main difference between PKE and a KEM is in the API. A KEM API does not allow to encrypt chosen messages, the message gets created for you.

A KEM offers three functions:

the public key *pk*

- (*ss*) = KEM_Decrypt(*sk*, *ct*), decrypts ciphertext *ct* to plaintext *ss* using private key *sk*

It is not hard to see that we can turn the Kyber PKE system into a KEM. Internally the Kyber KEM utilizes the Kyber PKE as we introduced it earlier.

Kyber Security

In its core, Kyber is essentially a Lyubashevsky, Peikert, Regev (LPR) encryption system within a module-learning-with-errors (MLWE) setting. The original LPR schemes were defined in LWE and Ring LWE settings. The difference between those is simple:

- **LWE** works with vectors of integers
- **RING LWE** works with polynomials
- **Module LWE** works with vectors of polynomials

While vectors of polynomials are certainly unintuitive to work with, they allow for much better speed/size/security tradeoffs.

MLWE (and therefore Kyber's) security is actually based on a lattice problem, namely the shortest vector problem (SVP). So recovering a message from a ciphertext (or a private from a public key), should be as hard as solving a large SVP instance. This should be computationally infeasible, even for a quantum computer. The reduction from Kyber to MLWE to SVP is non trivial, but well documented. If you are interested in the details I recommend reading Kyber's NIST submission and following the citations.

The main advantage of lattice-based schemes such as Kyber is that they are very fast, especially compared to other post-quantum systems. A disadvantage is that it is not fully clear if the additional structure of the underlying module lattices can be used to formulate better attacks on the specific SVP instance.

Conclusion

Kyber is a very fast lattice-based PKE, standardized as KEM. Its keys are bigger than those of pre-quantum schemes, but small enough to be used in real-world systems. The security of Kyber is based on the hardness of the MLWE problem which is in turn based on the hardness of the SVP problem. This makes Kyber an interesting candidate for many PQC applications.

#kyber #pqc #lattice #crypto #rugo

Approachable Cryptography

Ruben Gonzalez

Im a Ph.D. student in the field of applied cryptography at Radboud University.