Post Quantum Cryptography

Brian Ton

October 18, 2025

Contents

1 Some Preliminaries

This document is a collection of notes on post quantum cryptography. It's mainly for my own benefit, but I hope it's useful to others as well. It focuses on the math behind the algorithms and why they work. If you want a more intuitive understanding, I recommend checking out the YouTube video series by Chalk Talk. For a more in-depth treatment, you can also check out this YouTube video by Another Roof.

I was inspired to write this document after Sanyukta walked up to me when I was trying to understand [7] and told me to send it to her (but I didn't understand what I was reading). This document is a synthesis of everything that I read about to understand what was going on. The references I mainly pulled from are [7], [3], [12], and [6]. The appendix at the end contains some math background that I think are fairly necessary for understanding what's going on (mainly abstract algebra and number theory, and how they relate to classical cryptography). My main goal is to explain it at a level that's accessible to someone who has a good understanding of classical cryptography and the math behind it, but doesn't need every single detail.

2 The Main Threats

It's often said that quantum computers will "break" encryption (e.g. this article). But how true is this exactly? To answer this question, we will need to look at two main algorithms: **Grover's Algorithm** and **Shor's Algorithm** to evaluate what the risks are.

As we'll see in this section, the idea that quantum computers are catastrophic for current "classical" encryption is a partial truth: the risks for symmetric encryption like AES are quite manageable, while asymmetric encryption (in the flavor of RSA and Diffie-Hellman) is essentially broken wide open.

2.1 Symmetric Encryption

At a theoretical level, the major threat to symmetric cryptography is Grover's algorithm. Grover's algorithm helps us solve the problem of function inversion, namely:

Problem 2.1. *Given a (black-box) function $f : X \to Y$ and an output $y \in Y$, find an input $x \in X$ such that $f(x) = y$.*

I'll leave out the quantum details of how it actually gets done (not too relevant for our purposes). Handwaving a bit, you can imagine letting f be something like AES, we can launch a key retrieval attack by simply picking a message m, computing $c = f(m)$, and then using Grover's algorithm to find the key k such that $f(k) = c$.

But of course you can do that with a classical computer as well! It's just going to take a really long time. In particular, such a brute force attack would take $O(n)$ queries to f to find the key. The benefit of Grover's algorithm is that it takes only $O(\sqrt{n})$ queries to find the key, which is unfortunately a fairly significant improvement.

Luckily, we can very easily mitigate against this attack by simply using a larger key size. Since Grover's algorithm cuts our key size effectively in half (to understand why, notice that $\sqrt{2^{256}} = 2^{128}$), so we simply need to **double the key size** to get the same level of security! This is why symmetric cryptography has not been a major concern for quantum computers.

As a side note, it was shown in [2] that Grover's algorithm is asymptotically optimal for this problem, so anyone wanting to break AES (or any other symmetric encryption scheme) would need to be more clever, probably by doing some kind of cryptanalysis.

2.2 Asymmetric Encryption

Unfortunately, asymmetric encryption is not so lucky. In particular, Shor's algorithm is able to solve the problems of factoring large integers and computing discrete logarithms in polynomial time. This is a significant threat to RSA and Diffie-Hellman. Peter Shor's work presented here won him the 1999 Gödel Prize (along with many other awards).

2.2.1 RSA

The core of breaking RSA lies in factoring large integers. A small note on RSA is in the appendix (7.2), but let's mention it here for completeness. For RSA, one picks two prime numbers p and q, and computes $n = pq$. The public key is then (n, e) where e is a public exponent (e.g. 3 or 65537) and d is the private exponent such that $ed \equiv 1 \mod \phi(n)$ where $\phi(n) = (p-1)(q-1)$ is the Euler totient function. The private key is then (n, d). The message is then encrypted as $m^e \mod n$ and decrypted as $c^d \mod n$.

For an attacker, the goal is to compute d given the public information (n, e). If they know $\phi(n)$, then computing d can be easily done in polynomial time via the extended Euclidean algorithm. Thus, the security of RSA is based on the hardness of computing $\phi(n)$. The best known way to compute $\phi(n)$ is our earlier formula $\phi(n) = (p-1)(q-1)$, so we really want p and q to compute d, and hence the best hope for the attacker is to factor $n = pq$. On a classical computer, the best-known algorithms run in exponential time [4]. Shor's algorithm will be much better than that!

2.2.2 Shor's Algorithm

How does Shor's algorithm work? At a high level, the steps are as follows (pulling from [10]). Let N be an odd integer. We will output two nontrivial factors of N.

1. Pick a random integer a such that $1 < a < N$.
2. Compute $x = \gcd(a, N)$.
3. If $x \neq 1$, then x is a nontrivial factor of N, and output $(x, N/x)$.
4. Otherwise, use a quantum subroutine to compute the order (7.4) m of a modulo N
5. If m is odd, then go back to Step 1.
6. Otherwise, compute $y = \gcd(N, a^{m/2} + 1)$.
7. If $y = 1$ or $y = N$, then go back to Step 1.
8. Otherwise, y is a nontrivial factor of N, and output $(y, N/y)$.

The key math idea here is this: If we hit step 4, then a and N are coprime, so $a \in \mathbb{Z}_N^*$, the multiplicative group of integers modulo N. In particular, by a group theory fact (7.1), m will be finite, so that $a^m \equiv 1 \mod N$, so N divides $a^m - 1$. If m is even, then $a^m - 1 = (a^{m/2} + 1)(a^{m/2} - 1)$, so N divides $(a^{m/2} + 1)(a^{m/2} - 1)$, and we can try to capture a divisor of N by taking the gcd (which can be done efficiently using the Euclidean algorithm). The quantum subroutine here is also the true genius of the algorithm.

Overall, Shor's algorithm factors numbers in $O((\log N)^2(\log\log N)(\log\log\log N))$ time, an incredible speedup from the best known classical algorithm (which runs in exponential time). **This is why most people consider RSA broken by quantum computers.** Still, the largest number we've factored so far with Shor's algorithm is 21, something even I could have done in second grade.

2.2.3 Diffie-Hellman

Diffie-Hellman is based on the discrete logarithm problem. In particular, let G be a finite cyclic group (7.6) and a be a generator of G. Then, the discrete logarithm problem is to compute $\log_a b$ given a and b (namely, the smallest n such that $a^n = b$). It is a hard problem to solve in general, and no known efficient algorithm is known to exist for a classical computer. But an algorithm similar to 2.2.2 can be used to solve it efficiently on a quantum computer.

2.2.4 Hidden Subgroup Problem

We will describe the hidden subgroup problem.

Definition 2.1. Let G be a group (7.1) and H be a subgroup of G (7.2). Let X be a set, and let $f : G \to X$ be a function. We say that f **hides** H if for all $g_1, g_2 \in G$, $f(g_1) = f(g_2)$ if and only if $g_1 = g_2 h$ for some $h \in H$. Equivalently (if you know what cosets are), f hides H if for all $g_1, g_2 \in G$, $f(g_1) = f(g_2)$ if and only if $g_1 H = g_2 H$.

The hidden subgroup problem is then to use evaluation queries to f to determine a generating set (7.5) for H.

2.2.5 Shor's Algorithm for Discrete Logarithm

Shor's algorithm for the discrete logarithm solves the hidden subgroup problem for a specific function. Namely, suppose we have a finite group G of prime order n, generator g, and element x that we want to find the discrete log of. We can define the function $f : \mathbb{Z}_n \times \mathbb{Z}_n \to G$ where $f(a, b) = x^a g^{-b}$. Since g is a generator, $f(a, b) = g^{a \log_g x - b}$, and one checks that f hides the subgroup $H = \{m(1, \log_g x) : m \in \mathbb{Z}_n\}$. The way once checks this is to first prove that $f(ac, bd) = f(a, b)f(c, d)$ for all $a, b, c, d \in \mathbb{Z}_n$, then show that $f(a, b) = 1$ if and only if $(a, b) \in H$. Indeed, once we find the generator of this H, which is $(1, \log_g x)$, we obtain the discrete log of x by looking in the second coordinate! And this is exactly what Shor's algorithm does (with some quantum superposition magic). There exist general quantum algorithms to solve the hidden subgroup problem for any finite abelian group using representation theory.

What does this mean, though? **It means that we can break Diffie-Hellman in polynomial time using a quantum computer!**

3 Fun with Lattices

Now that we've seen how Shor's algorithm breaks both factoring and discrete logs, let's see how we can rebuild security from the ground up, starting with lattices. This section is an introduction to what lattices are and the core computational problems that make them useful for cryptography. If you want to skip to the practical stuff, you can skip this section to 4 (though this section is fairly foundational).

3.1 Lattices

The main object of study for lattice-based cryptography is of course the lattice. Intuitively, a lattice is a sparse set of points within a grid. We'll consider integer lattices, which are a restricted case.

Definition 3.1. A lattice $L \subseteq \mathbb{R}^n$ is the set of all integer linear combinations of a set of basis vectors $\mathcal{B} = \{b_1, \ldots, b_n\} \subseteq \mathbb{R}^n$. That is, $L = L(\mathcal{B}) = \{\sum_{i=1}^n c_i b_i : c_i \in \mathbb{Z}\}$.

In other words, a lattice is a set of points that is closed under addition and (integer) scalar multiplication. The most common example of a lattice is the integer lattice, $\mathbb{Z}^n \subseteq \mathbb{R}^n$. This gives us a visualization of a lattice as a grid of points, like the picture of the lattice below:

Since lattices are nice subsets of vector spaces, we can use a lot of the regular linear algebra tools to understand them. In particular, let $\mathcal{B} = \{b_1, \ldots, b_n\}$ be a basis for $L(\mathcal{B})$. Then, for any vector $v \in L(\mathcal{B})$, we

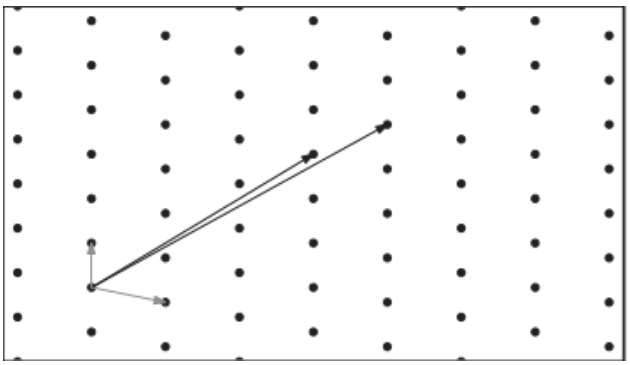

Figure 1: A lattice spanned by two basis vectors (in light gray). Stolen from [3].

can write $v = \sum_{i=1}^{n} c_i b_i$ for some $c_i \in \mathbb{Z}$. And in particular, any vector $v \in L(\mathcal{B})$ can be written as $v = Bc$ for some $c \in \mathbb{Z}^n$, where $B = [b_1 \dots b_n]$ is the matrix whose columns are the basis vectors. That is, we can represent a movement within a lattice as a matrix multiplication:

$$v = \begin{bmatrix} b_{1,1} & \cdots & b_{n,1} \\ \vdots & \ddots & \vdots \\ b_{1,n} & \cdots & b_{n,n} \end{bmatrix} \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix}$$

And note that any matrix multiplication of this form is also indeed in the lattice as well.

3.2 Hard Lattice Problems

There are two main hard lattice problems that we'll consider: the Shortest Vector Problem (SVP) and the Closest Vector Problem (CVP). They are essentially what they sound like.

Definition 3.2. The Shortest Vector Problem (SVP) is to, given a lattice basis $\mathcal{B}$, find a shortest non-zero vector in the corresponding lattice $L(\mathcal{B})$.

Definition 3.3. The Closest Vector Problem (CVP) is to, given a lattice basis $\mathcal{B}$ and a target vector t, find a vector $v \in L(\mathcal{B})$ such that $||v - t||$ is minimized (i.e. v is the closest vector to t in $L(\mathcal{B})$).

It is known that both of these problems are NP-hard (although under potentially random reductions for SVP under certain norms). This means that it's probably a good idea to base our cryptography on them.

3.3 Learning with Errors

The core of the NIST-selected lattice-based PQC algorithms is the Learning with Errors problem. Here, we discuss the classic version:

Problem 3.1. *Fix $s \in \mathbb{Z}_q^n$ and $A \in \mathbb{Z}_q^{n \times m}$. Let $e \in \mathbb{Z}_q^m$ be a random error vector. Compute $b = As + e \mod q$. The Learning with Errors problem (LWE) is to find s given A and b.*

The picture of the matrix multiplication looks like:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix} \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_m \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_m \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

Intuitively, if e were 0, then the problem would be very easy! Just calculate $s = A^{-1}b$. But, since e introduces some "noise", our traditional linear algebra techniques don't apply. Using our note from 3.1, consider the lattice

$$L(A) = \{y \in \mathbb{Z}^m : y = Ax \mod q \text{ for some } x \in \mathbb{Z}^n\}$$

We can see that if e is small enough, then the Learning with Errors problem asks us to find the closest vector $s \in L(A)$ to b, which we know should be hard.

More formally, Oded Regev proved in [9] that the average-case version of LWE is NP-hard by showing a reduction from the worst-case version of a variant of SVP to the average-case version of the Learning with Errors problem. Namely, if we can solve the average-case version of LWE, then we can solve the worst-case version of (a variant of) SVP, which is known to be NP-hard. He won the 2018 Gödel Prize for this work.

3.3.1 Module Learning with Errors

I won't go too into the details of module learning with errors here, but it's a generalization of the Learning with Errors problem to general modules (rather than just ones that are a subset of $\mathbb{Z}^n$).

Using our terminology from 7.3, define a module lattice as follows:

Definition 3.4. Let R be an integral domain (7.10) with field of fractions K. Let V be a vector space over K. Then, a module lattice M over R is a finitely generated submodule of V.

If that makes no sense to you, don't worry too much about it. All we care about for the next section is that it basically means that instead of matrices with numbers like in the previous section, we can now deal with matrices of polynomials.

Why do we care? Well, it gives us huge gains in efficiency. We can do stuff like the so-called "Number Theoretic Transform" when working with polynomials. And additionally, it allows us to shrink key sizes by a huge amount. While this does potentially open us up to attacks that use the structure of R or M, no such attacks are known.

3.4 Practical State of Lattice-Based Cryptography

Lattice-based cryptography was made part of the NIST standard for Key Encapsulation and Digital Signatures with FIPS 203 and FIPS 204, respectively. It's also now already implemented in our favorite cryptography libraries! Well, kinda. For Java, you can update to Java 24 for access to JEP 496 and JEP 497, which add native Module-Lattice-based interfaces to the Java Security engine. And BouncyCastle got implementations of module-lattice crypto in BC 1.79. In the next sections, we'll talk about the NIST-selected algorithms that use lattices: Kyber and Dilithium.

4 Kyber

Kyber is the NIST-selected PQC algorithm for Key Encapsulation. It takes the place of something like RSA. As a side note, I believe the name is a reference to Star Wars, but no one online seems to mention anything about it. This section is essentially based on [12] and [6].

4.1 Public Key Encryption

The Kyber PKE is the building block of this entire section. It allows us to encrypt messages of length equal to 32 bytes. In particular, it is **not approved** for use in a standalone fashion for arbitrary-length messages. It serves only as a collection of subroutines for the KEM (see section 5 of [12]), which we'll discuss in 4.2.

For this section and the next, let R_q be the ring of polynomials $\mathbb{Z}_q[X]/(X^n+1)$ for a prime number q. These are pretty much polynomials with coefficients in $\{0, ..., q-1\}$. For instance, if $q = 5$ and $n = 3$, then an element of R_q may look like $4x^3 + 2x + 1$. Here, n and q are going to be security parameters, that we will specify later.

4.1.1 Kyber Keys

A secret key in Kyber is a vector of k "small" polynomials $s \in R_q^k$. In particular, small means small in terms of coefficients, with each one far less than $\frac{q}{2}$.

A public key in Kyber is a pair (A, t), where A is a $k \times k$ square matrix of *random* polynomials $A \in R_q^{k \times k}$, and t is a k-dimensional vector of *random* polynomials $t \in R_q^k$. t is derived from A and s via the following equation:

$$t = As + e$$

where e is a random small error vector.

Notice that it's hard to derive s from A and t, as one would need to solve the Learning With Errors problem discussed in 3.3.

4.1.2 Encryption

How do we now encrypt a message? Let's let m be a 32-byte message. Our first step is to encode it as a polynomial by making each bit a coefficient of a polynomial m_b. As an example, the encoding of the string 1011 (assuming it's left-padded by 0's) is the polynomial $m_b = 1x^3 + 0x^2 + 1x + 1 = x^3 + x + 1$. We then generate a new error vector $e_1 \in R_q^k$, randomizer polynomial vector $r \in R_q^k$, and error polynomial $e_2 \in R_q$, all in a secure random manner (with small coefficients similar to how we generate s).

Before encrypting, we need to scale the coefficients of the polynomial m_b by a factor of $\lfloor \frac{q}{2} \rceil$ (i.e. the nearest integer to $\frac{q}{2}$) to get $m = \lfloor \frac{q}{2} \rceil m_b$. Then, we compute the vectors

$$\begin{cases} u = A^\top r + e_1 \\ v = t^\top r + e_2 + m \end{cases}$$

The ciphertext is then the combination of the vector u along with the polynomial v in the pair (u, v).

4.1.3 Decryption

Now, given the private key s and cipertext (u, v), we first compute a noisy result $m' = v - s^\top u$. Expanding it out using some matrix math, we get:

$$\begin{aligned} m' = v - s^\top u &= t^\top r + e_2 + m - s^\top (A^\top r + e_1) \\ &= (As + e)^\top r + e_2 + m - s^\top A^\top r - s^\top e_1 \\ &= (As)^\top r + e^\top r + e_2 + m - s^\top A^\top r - s^\top e_1 \\ &= s^\top A^\top r + e^\top r + e_2 + m - s^\top A^\top r - s^\top e_1 \\ &= e^\top r + e_2 + m - s^\top e_1 \end{aligned}$$

Now, since we made the coefficients of m big and the polynomials in e_1, e_2, and e small, we can decode m' to get m by going through each of the coefficients of m' and checking whether it is closer to 0 or $\frac{q}{2}$. If it's closer to 0, then we decode the value as 0. Otherwise, we decode the value as 1. For instance, if $m' = 2x^3 + 3x^2 + 4x + 5$ with $q = 11$, then we decode the value as 0011.

If you notice, there is a chance that m' is not equal to m. We mitigate this via correct choices of the security parameters n and q, as discussed in 4.3.

4.1.4 Efficiency and Compression

There are a few quick ways to make Kyber more efficient than the naive approach.

Instead of storing the entire $A \in R_q^{k \times k}$ in the public key and, we simply need to store a 32-byte seed the 32-byte seed used to generate it, which will save quite a lot of bytes.

In the real world, we also apply compression to the ciphertext to hopefully save some bits. To do this, first fix a number of bits d that we want to use. Then, take q and assign each value $v \in \{0, ..., q-1\}$ to a value $v' = \lfloor \frac{v}{q} \cdot 2^d \rceil \mod q$. Essentially, we're scaling down the full range of values to the range of values that we can represent with d bits. Then, to decompress, compute this in reverse. The intuition is that since we only care about "big" and "small" values when encrypting / decrypting, it's ok to throw away a lot of the granularity when encrypting / decrypting. It can save $\approx \frac{1}{3}$ of the total space of the ciphertext. Again, d is a security parameter we can adjust (see 4.3), and it also does affect the security of the scheme.

4.2 Turning The PKE into a KEM

So the above gives us an IND-CPA scheme for public key encryption. At a high level, that means that given two plaintexts and one encrypted output, an attacker wouldn't be able to tell which plaintext corresponded to the output. This is good!

A KEM has 3 basic algorithms that we need to create: key generation, encapsulation, and decapsulation (and this is what's available via cryptography libraries).

- Key Generation: $(pk, sk) = \text{KeyGen}()$. Takes no inputs and returns a public key and a secret key.
- Encapsulation: $(k, c) = \text{Encaps}(pk)$. Randomly generates a key k and returns it along with its encapsulation (encryption) c.
- Decapsulation: $k' = \text{Decaps}(sk, c)$. Takes the secret key sk and the encapsulation c and returns a decapsulation of c.

An initial plan to turn our PKE into a KEM is to just have Alice generate a symmetric key, encrypt it with our public key, and send it over to Bob. To decapsulate, Bob would just decrypt and then send whatever message he wanted back (after encrypting with the symmetric key).

But the above scheme has a problem! A bad actor Charlie can intercept a legitimate encapsulation, try to modify it a bit, and send it to Bob. When Bob decrypts it, he won't get the original key that Alice sent! Charlie can then observe what Bob returns, potentially learning stuff about the original symmetric key, the secret key, or both. This is called a **chosen ciphertext attack**.

Via something called the **Fujisaki-Okamoto transform**, you can turn the PKE into a KEM. In particular, it turns our IND-CPA PKE into an IND-CCA2 KEM. The basic idea is this: during decapsulation, take the outputted key, then re-encrypt it with the public key. If it doesn't encrypt back to the same ciphertext, then implicitly reject the key by returning a pseudorandom string.

At a high level:

- Key generation: Generate the PKE keys (ek, dk). Generate a random 32-byte value z. The encapsulation key is the encryption key ek, and the decapsulation key is the concatenation of the decryption key dk, the encryption key ek, the (SHA3-256) hash of the encryption key $H(ek)$, and the random 32-byte value z.
- Encapsulation: Generate a random 32-byte string m. Then, compute $(K, r) = \text{SHA3-512}(m || H(ek))$. Encrypt K with the public key ek to get c (and use r as the random seed). Then, return K as the symmetric key and c as the encapsulation.

- Decapsulation: Decrypt c with the secret key sk to get m'. Recompute $(K', r') = \text{SHA3-512}(m||H(ek))$. If $K' \neq K$, implicitly reject the ciphertext by returning the pseudo-random string $\text{SHAKE256}(z||c)$. Otherwise, return m' as the symmetric key.

For the most amount of detail, see section 7 of [12] (specifically, Algorithm 16, 17, and 18).

4.3 Security Parameters

A Kyber parameter set is a tuple $(n, k, q, \eta_1, \eta_2, d_u, d_v)$, where n is the degree of the polynomials (essentially controlling the length of the message), k is the number of polynomials in the secret key, q is the modulus, η_1 and η_2 control how big "small" polynomials can be (by specifying the distribution they come from), d_u is the number of bits to use for the compression of the ciphertext, and d_v is the number of bits to use for the compression of the plaintext. Here is a table of the approved parameter sets, and some details about them, lifted from [12]:

Parameter Set	n	k	q	η_1	η_2	d_u	d_v	Decapsulation Failure Rate
Kyber 512	256	2	3229	3	2	10	4	2^{-139}
Kyber 768	256	3	3229	2	2	10	4	2^{-164}
Kyber 1024	256	4	3229	2	2	11	4	2^{-174}

That's a pretty good decapsulation failure rate! You're far more likely to have a UUIDv4 collision than a decapsulation failure.

5 Dilithium

Now that we know about Kyber, we can move on to Dilithium (a Star Trek reference as well I think). Dilithium is formalized in FIPS 204 and it is also a module-lattice-based cryptosystem, based on the Fiat-Shamir with Aborts cryptosystem [8]. However, in contrast to Kyber, Dilithium does not do any kind of encryption/decryption. Instead, it is a digital signature scheme. What does that mean? At a high level, it allows a sender **sign** a message with their private key, and anyone can **verify** the signature with the sender's public key. These sorts of schemes allow us to be sure that the message was indeed sent by the sender and not by a malicious actor. Less stuff online has been written about Dilithium, so this section is most directly based on the FIPS standard [11].

5.1 Key Generation

A Dilithium key pair is a pair (sk, pk), where sk is the secret key and pk is the public key. Similarly to Kyber, again let $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ and pick $sk \in R_q^\ell$ to be a vector of ℓ small polynomials. For the public key, we pick $A \in R_q^{k \times \ell}$ to be a random matrix of polynomials and $e \in R_q^k$ to be a small error vector. Then, compute $t = As + e$ as before. The public key is then the pair $pk = (A, t)$.

Something that may be of note here is that unlike with RSA (whose keys can be used for signing and encryption), Kyber and Dilithium keys are incompatible!

5.2 Signing

To sign a message m, we intuitively want something like $H(m) + s$ to combine something about the message, along with something from our secret key. We do this as follows.

First, the signer samples a random vector $y \in R_q^\ell$, and then computes $w = Ay$. Round w to w_1 by discarding the lower γ_2 bits (i.e. $w_1 = \text{HighBits}(w, \gamma_2)$). The signer then computes the challenge $c = H(w_1||\mu)$, where $\mu = H(H(pk)||m)$ is a message representative (for stronger unforgeability) and H is SHA3, and calculates a candidate $z = y + cs$.

We then run some checks on z (see 5.4). If the checks fail, then restart the signing process. Otherwise, the signature is (z, c).

Note that this re-sampling process for z until it meets certain conditions is called "rejection sampling". The choice of security parameters influences the expected number of times that we must re-sample, but with the approved ones, it should be less than 6. There are additional details here for the full signing process, like compressing t and doing some other stuff like that, similar to 4.1.4, but I'll omit the details since it essentially is the same (and a second discussion doesn't add much since it makes stuff more complicated with the high bits stuff already). Similarly, we don't really use c directly, we use it as the seed to a pseudo-random number generator to pull out 60 distinct positions to sample exactly 60 non-zero positions to assign non-zero coefficients of a new polynomial $\bar{c}$ to (but the details of the sampling really don't matter, so I left them out).

The rounding (HighBits) step ensures that both sides interpret small coefficient differences consistently. Since the amount of noise with cs is bounded, it keeps verification deterministic (as we'll see).

5.3 Verification

To verify, one does essentially the reverse process. Firstly, the verifier takes z and computes $w' = \text{HighBits}(Az - ct, \gamma_2)$. Then, the verifier again computes $\mu = H(H(pk)||m))$ and $c' = H(w'||\mu)$. The verifier then verifies $c' = c$.

To see that it is a valid signing scheme, we expand

$$\begin{aligned} Az - ct &= A(y + cs) - c(As + e) \\ &= Ay + Acs - cAs - ce \\ &= Ay - ce \\ &\approx Ay = w \approx w_1 \end{aligned}$$

In particular, we see that the second check on z mentioned in 5.4 ensures that

$$w' = \text{HighBits}(Az - ct, \gamma_2) = \text{HighBits}(Ay, \gamma_2) = w_1$$

hence $c = c'$, so our verification is successful.

5.4 Why Rejection Sampling?

During signing, we check the following for z:

1. All of z's coefficients are less than $\gamma_1 - \beta$
2. All lower bits of $Az - ct$ are smaller than $\gamma_2 - \beta$

where β is the maximum possible coefficient of cs.

Here, the goal of this sampling process is to prevent side-channel attacks where an attacker can leak parts of the values of s or e via sampling *many* signatures and seeing their distribution. In particular, we are ensuring that z only comes from a distribution that doesn't leak any values. The first protects against leaks of s as the "center" of the distribution of z, and the second protects against the lower bits of $Ay - ct$ being too large, which can cause failures in decryption (since then the computed w' may not be equal to w_1), and an attacker may be able to learn things about e.

5.5 Security Parameters

This is an abridged version of the security parameters table from section 4 of [11] (using the parameters we've mentioned in the above description):

Parameter Set	q	γ_1	γ_2	(k, ℓ)	β	Expected Repetitions	Claimed Security Strength
ML-DSA-44	8380417	2^{17}	$(q-1)/88$	$(4, 4)$	78	4.25	Category 2
ML-DSA-65	8380417	2^{19}	$(q-1)/32$	$(6, 5)$	196	5.1	Category 3
ML-DSA-87	8380417	2^{19}	$(q-1)/32$	$(8, 7)$	120	3.85	Category 5

where the security strength is according to section 4.A.5 of the NIST PQC evaluation criteria. Here we note that indeed, we need, on average, around 4-6 repetitions of the rejection sampling on each signature.

6 Conclusion

So we've just taken a trip through a whole lot! We first looked at the failures of classical encryption with respect to quantum computers. Then, took a look at a basic building block of PQC: lattices. Finally, we examined Kyber and Dilithium, which utilize lattices to construct KEM and DSA primitives. I think one key takeaway here is that for these PQC schemes, small amounts of randomness can turn easy problems into very hard ones. In short, post-quantum cryptography shifts from number theory to geometry — hardness now lies in structure, not size. Hopefully, cryptography is safe for a while, until the next type of computer is invented! DNA computing, anyone? ☺

7 Appendix

7.1 Math Notation

Here is some math notation that I used throughout this document.

- $\mathbb{Z}$ is the set of integers.
- $\mathbb{Q}$ is the set of rational numbers.
- $\mathbb{R}$ is the set of real numbers.
- $\{0,1\}^n$ is the set of all n-bit strings of 0's and 1's.

7.2 Group Theory

This section is just a dumping ground for some basic group theory plus useful results; for a more thorough treatment as it pertains to cryptography, see [1] or a book on abstract algebra like [5]. The main definition of interest for this subsection is the group.

Definition 7.1. A group is a set G equipped with a binary operation $\cdot$ that satisfies the following axioms:

1. (Closure) For all $a, b \in G$, the element $a \cdot b \in G$.
2. (Associativity) For all $a, b, c \in G$, the equation $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ holds.
3. (Identity) There exists an element $e \in G$ such that for all $a \in G$, the equation $e \cdot a = a \cdot e = a$ holds.
4. (Inverses) For each $a \in G$, there exists an element $b \in G$ such that $a \cdot b = b \cdot a = e$.

Some important examples of groups are:

- The integers $\mathbb{Z}$ under addition, often denoted $(\mathbb{Z}, +)$.
- The integers modulo n, $\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$ under addition modulo n, often denoted $(\mathbb{Z}_n, +_n)$.
- The multiplicative group of integers modulo n, $\mathbb{Z}_n^* = \{x : \gcd(x, n) = 1\}$ under multiplication modulo n, often denoted $(\mathbb{Z}_n^*, \cdot_n)$. In particular, $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$.
- The set of all $n \times n$ invertible matrices under matrix multiplication.

The first three examples differ from the last in an important way. Namely, their group operation is commutative. For instance, $a + b = b + a$ for any integers $a, b \in \mathbb{Z}$ and similarly for $\mathbb{Z}_n$. We call such groups **abelian** groups. The last example is not abelian, as matrix multiplication is not commutative in general (see here for an example).

As another side note, the group $\mathbb{Z}_n^*$ is the main backbone of RSA encryption. In particular, you encode your message as an integer $m \in \mathbb{Z}_n$, and then you encrypt it as $m^e \mod n$ for some public exponent e with (n, e) becoming your public key. The private key, d, is the inverse element of e in $\mathbb{Z}_n^*$. That is, $ed \equiv 1 \mod n$. With a bit of handwaving, we can see that $m^{ed} \mod n = m^1 \mod n = m$, i.e. we can decrypt the message!

7.2.1 Other useful definitions and results

Here are some other useful definitions and results relating to groups that are useful for the rest of this document. For proofs, see [5].

Definition 7.2. A **subgroup** H of a group G is a subset $H \subseteq G$ that is itself a group under the operation of G.

Definition 7.3. The **order** of a group G, denoted $|G|$, is the number of elements in G.

Definition 7.4. The **order** of an element $a \in G$, denoted $|a|$, is the smallest positive integer n such that $a^n = e$.

Note that there can be elements and groups with infinite order. For instance, the group of integers under addition has infinite order, and the element 2 has infinite order. There's also a nice theorem that relates the two notions of order:

Theorem 7.1 (Lagrange's Theorem)**.** *Let G be a finite group of order n. Then, for any element $a \in G$, $|a|$ divides $|G|$.*

Definition 7.5. Let G be a group. Let $S \subseteq G$ be a subset. We say that S is a **generating set** for G if every element of G can be written as a finite product of elements of S and their inverses.

Definition 7.6. A group G is **cyclic** if there exists an element $a \in G$ such that every element of G can be written as a^n for some integer n. The element a is called a **generator** of the group. Equivalently, G is cyclic if it has a generating set of size 1.

Cyclic groups are the backbone of cryptography! In particular, Diffie-Hellman Key Exchange is based on the fact that the group of integers modulo a prime number p, $\mathbb{Z}_p^*$, is cyclic (or an elliptic curve group with a prime order). And in particular, its security is based on the discrete logarithm problem.

Definition 7.7. Let G be a finite cyclic group with generator a and order n. The **discrete logarithm** of an element $b \in G$ with respect to a generator $a \in G$ is the smallest positive integer n such that $a^n = b$. This is denoted $\log_a b$.

The discrete logarithm problem is then to compute $\log_a b$ given a and b. It is a hard problem to solve in general, and no known efficient algorithm is known to exist for a classical computer. However, see 2.2.5 for a discussion on how it can be solved efficiently on a quantum computer.

7.3 Ring Theory

A ring is a group with more stuff! The best picture in your head is to think of the integers. You can add them, subtract them, and multiply them. However, you can't divide them. Rings are the backbone of Post Quantum Cryptography.

Definition 7.8. A (commutative) **ring** is a set R equipped with two binary operations $\cdot$ and $+$ that satisfy the following axioms:

1. $(R, +)$ is an abelian group.
2. $(R, \cdot)$ satisfies commutativity, associativity, and has an identity element.
3. Distributivity: for all $a, b, c \in R$, $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$.

Some important examples of rings are:

- The integers $\mathbb{Z}$.
- The integers modulo n, $\mathbb{Z}_n$, under addition and multiplication modulo n.

There are two important classes of rings that we will use in this document: fields and integral domains.

Definition 7.9. Let R be a ring. We say that R is a **field** if every non-zero element has a multiplicative inverse. That is, for all $a \in R$, $a \neq 0$, there exists $b \in R$ such that $a \cdot b = b \cdot a = 1$.

Definition 7.10. Let R be a ring. We say that R is an **integral domain** if it is a commutative ring with no zero divisors. That is, for all $a, b \in R$, $a \cdot b = 0$ implies $a = 0$ or $b = 0$.

Note that all fields are integral domains, but not vice versa. Luckily, there's a nice theorem that relates the two:

Theorem 7.2. *Let D be an integral domain. Then, there exists a field F such that $D \subseteq F$ and F is the smallest field containing D. This field is called the* ***field of fractions*** *of D.*

The reasoning behind the name of fractions is that we can write F as essentially elements of the form $\frac{a}{b}$ where $a, b \in D$ and $b \neq 0$. Indeed, for $\mathbb{Z}$ (the set of integers), the field of fractions is $\mathbb{Q}$ (the set of all rational numbers).

We will use rings here to define the notion of a module, which is the main component of Kyber, one of the NIST-selected post-quantum algorithms (and the one mentioned in 3.3.1).

Definition 7.11. A **module** over a ring R is an abelian group M equipped with a binary operation $\cdot$ that satisfies the following axiom: for all $a, b \in R$ and $x, y \in M$, $a \cdot (x+y) = (a \cdot x) + (a \cdot y)$ and $(a+b) \cdot x = (a \cdot x) + (b \cdot x)$.

Namely, a module generalizes the notion of a vector space over a field to a ring. Indeed, if R is a field, then a module over R is just a vector space over R.

References

[1] Mihir Bellare. *Computational Number Theory*. 2018. URL: https://cseweb.ucsd.edu/~mihir/cse107/slides/s-cnt.pdf (visited on 10/08/2025).

[2] Charles H Bennett et al. "Strengths and weaknesses of quantum computing". In: *SIAM journal on Computing* 26.5 (1997), pp. 1510–1523.

[3] Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen. *Post-Quantum Cryptography*. Springer, 2009. ISBN: 978-3-540-88701-0. DOI: 10.1007/978-3-540-88702-7.

[4] J. P. Buhler, H. W. Lenstra, and Carl Pomerance. "Factoring integers with the number field sieve". In: *The development of the number field sieve*. Ed. by Arjen K. Lenstra and Hendrik W. Lenstra. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 50–94. ISBN: 978-3-540-47892-8.

[5] Joseph A Gallian. *Contemporary abstract algebra*. Ninth edition. Cengage Learning, 2017. ISBN: 978-1-305-65796-0.

[6] Ruben Gonzalez. *Kyber - How Does It Work?* Sept. 2021. URL: https://cryptopedia.dev/posts/kyber/.

[7] Nadia Heninger. *Post-Quantum Cryptography*. 2025. URL: https://cseweb.ucsd.edu/classes/fa25/cse207B-a/lectures/20-pqc.pdf (visited on 10/07/2025).

[8] Vadim Lyubashevsky. "Fiat-Shamir with Aborts: Applications to Lattice and Factoring-Based Signatures". In: *Advances in Cryptology – ASIACRYPT 2009*. Ed. by Mitsuru Matsui. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 598–616.

[9] Oded Regev. "On Lattices, Learning with Errors, Random Linear Codes, and Cryptography". In: *Journal of the ACM* 56.6 (2009), 34:1–34:40. DOI: 10.1145/1568318.1568324.

[10] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". In: *SIAM Journal on Computing* 26.5 (1997), pp. 1484–1509. DOI: 10.1137/S0097539795293172.

[11] National Institute of Standards and Technology. *Module-Lattice-Based Digital Signature Standard*. Tech. rep. Washington, D.C.: U.S. Department of Commerce, 2024. DOI: 10.6028/NIST.FIPS.204.

[12] National Institute of Standards and Technology. *Module-Lattice-Based Key-Encapsulation Mechanism Standard*. Tech. rep. Washington, D.C.: U.S. Department of Commerce, 2024. DOI: 10.6028/NIST.FIPS.203.